Universidad Zaragoza

Trabajo Fin de Grado

Sistema de localización con cámara RGBd basado en ICP sobre un entorno conocido

English title:

ICP-based RGBd camera location system on a known environment

Autor:

Ramiro Costa Arana

Director:

José Luis Villarroel Salcedo

Escuela de Ingeniería y Arquitectura
Zaragoza, Septiembre de 2019

RESUMEN

Sistema de localización con cámara RGBD basado en ICP sobre un entorno conocido

El objetivo de este proyecto es profundizar en el desarrollo de una aplicación capaz de determinar el camino seguido por una cámara RGBD dentro de un entorno conocido y a partir de una posición inicial también conocida.

La localización se ha realizado aplicando un algoritmo ICP (*Iterative Closest Point*), centrándonos en dos variantes de este algoritmo, el punto a punto y el punto a plano.

Este algoritmo se aplicará sobre una nube de puntos que conformará el mapa del entorno en estudio y otra nube de puntos captada por la cámara RGBD, que serán cada una de las capturas que forman la secuencia seguida por la cámara.

Los resultados obtenidos de las variantes del algoritmo ICP en determinadas situaciones conducirán a plantearnos si es necesario introducir un IMU o Unidad de Medición Inercial.

Para llevar a cabo todo lo anterior se construirá primeramente un mapa y posteriormente se grabará la secuencia que recogerá la trayectoria seguida sobre ese mapa.

La grabación de la secuencia se ha llevado a cabo mediante una aplicación de tiempo real que recoge la información de la cámara y genera una estructura de ficheros que podrá ser interpretada posteriormente.

La aplicación principal leerá de memoria esa secuencia y la procesará emulando una aplicación de tiempo real, analizando los datos en el orden temporal en que fueron recogidos.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS **9**

CAPÍTULO 1 ..... **11**

1. CONTEXTO Y ANTECEDENTES **11**

2. OBJETIVOS Y REQUISITOS **12**

2.1. Análisis de objetivos 12

3. ORGANIZACIÓN DE LA MEMORIA **14**

CAPÍTULO 2 ..... **15**

1. HARDWARE **15**

1.1. Cámara RGBD 15

2. SOFTWARE **17**

2.1. C++ 17

2.2. ROS (Robot Operating System) 17

2.3. PCL (Point Cloud Library) 18

2.4. RGBDSlam 19

2.5. RViz 20

CAPÍTULO 3 ..... **21**

1. ELECCIÓN DEL ENTORNO **21**

2. CONSTRUCCIÓN DEL MAPA **22**

CAPÍTULO 4 ..... **27**

1. GRABACIÓN PREVIA CON ROSBAG **27**

2. ESTRUCTURA DE DATOS DE LA APLICACIÓN **28**

3. DESCRIPCIÓN DE LA APLICACIÓN **30**

4. LECTURA DE LA CÁMARA **31**

CAPÍTULO 5 ..... **33**

1. DESCRIPCIÓN DE LA APLICACIÓN **33**

CAPÍTULO 6 ..... **35**

1. ESTRUCTURA DE DATOS **35**

2. DESCRIPCIÓN DE LA APLICACIÓN **37**

3. PROCESAMIENTO DEL MAPA **39**

4. LOCALIZAR NUBE DE PUNTOS **39**

4.1. Funcionamiento del algoritmo ICP 40

4.2. Implementación del algoritmo ICP 41

5. EJECUCIÓN DEL PROGRAMA **44**

CAPÍTULO 7 ..... **45**

1. SECUENCIA 1 ..... **45**

1.1. Análisis secuencia 1 46

1.2. Análisis de tiempos 47

2. SECUENCIA 2 ..... **49**

2.1. Análisis secuencia 2 49

2.2. Análisis de tiempos 50

3. SECUENCIA 3 ..... **52**

3.1. Análisis Secuencia 3 52

3.2. Análisis de tiempos 54

CAPÍTULO 8 ..... **55**

BIBLIOGRAFÍA **57**

ANEXOS **60**

ÍNDICE DE FIGURAS

Fig. 1.1. Ejemplo algoritmo ICP 13
Fig. 2.1. Partes del sensor ASUS Xtion PRO LIVE. [5] 16
Fig. 2.2. Ejemplo nube de puntos. 18
Fig. 2.3. Interfaz de la aplicación RGBDSlam. [10] 19
Fig. 2.4. Interfaz del programa RViz. 20
Fig. 3.1. Partes de la interfaz de RGBDSlam. [10] 23
Fig. 3.2. Mapa mal formado debido a una mala transformación entre *frames*. 24
Fig. 3.3. Mapa mal formado debido a giros y movimientos lineales al mismo tiempo. 24
Fig. 3.4. Mapa bien conformado siguiendo la parametrización descrita. 25
Fig. 3.5. Mapa creado con correcta parametrización. 26
Fig. 4.1. Ejemplo de secuencia. 28
Fig. 4.2. Sistema de ficheros utilizado en la aplicación de grabado de la secuencia 29
Fig. 4.3. Ejemplo de *parameters.txt* y *clouds_data.txt*. 29
Fig. 4.4. Diagrama de flujo de la aplicación *tfg_recorder*. 30
Fig. 4.5. Diagrama de flujo de la función que lee los mensajes almacenados provenientes de la cámara. 31
Fig. 5.1. Diagrama de flujo de la aplicación *tfg_set_origin*. 34
Fig. 6.1. Sistema de ficheros de *tfg_processor* 35
Fig. 6.2. Ejemplo de *origin.txt*, posiciones en matriz de rotación, r, y vector posición, p. 36
Fig. 6.3.a Diagrama de flujo de *tfg_processor*. 37
Fig. 6.3.b Diagrama de flujo de *tfg_processor*. 38
Fig. 6.4. Diagrama de flujo del proceso de localización de nube. 41
Fig. 6.5. Diagrama de flujo de la función de llamada al ICP. 42
Fig. 6.6. Mensaje de ayuda de *tfg_processor*. 44
Fig. 7.1. Mapa de la secuencia 1. 45
Fig. 7.2. Procesamiento completo de la secuencia 1. 46
Fig. 7.3. Procesamiento completo de la secuencia 2. 49
Fig. 7.4. Mapa utilizado en la secuencia 3. 52
Fig. 7.5. Procesamiento completo de la secuencia 3. 53

CAPÍTULO 1

Introducción

1. CONTEXTO Y ANTECEDENTES

Tanto las nubes de puntos como las imágenes nos ofrecen información geométrica del mundo que nos rodea, pero sus similitudes no van mucho más allá.

Mientras en las imágenes tenemos una representación bidimensional de una escena en una cuadrícula fija, en las nubes de puntos tenemos una colección de puntos tridimensionales no ordenados en un sistema coordenado unificado, es decir, poseemos información espacial en 3 dimensiones sobre una escena. [1]

En los últimos años, debido a la evolución y abaratamiento de sensores capaces de capturar color y profundidad, han aparecido nuevas técnicas de procesamiento de estas colecciones. Nos referimos más concretamente a las técnicas o algoritmos conocidos como registro de nubes de puntos, que se basan en el alineamiento de éstas cuando son capturadas en diferentes estampas temporales.

Este alineamiento se basa en el cálculo de la transformación que mapea de manera óptima dos nubes de puntos. Existe una gran variedad de algoritmos de registro, desde los que modelan las transformaciones usando 6 grados de libertad hasta los que hacen frente a objetos articulados que cambian de forma con el tiempo, pasando también por los que hacen una aproximación geométrica inicial o los que calculan la transformación que registra dos nubes de puntos de manera precisa. [2]

Estos algoritmos de registro son usados en diferentes campos y aplicaciones, como por ejemplo escaneo de objetos 3D, mapeo en 3D, localización en 3D o detección de cuerpos humanos. Muchas de estas aplicaciones utilizan algoritmos de registro basados en la clasificación que se ha hecho anteriormente, como por ejemplo el valor singular de descomposición (*SVD, Singular Value Decomposition*), análisis de componentes principales (*PCA, Principal Component Analysis*) o una aproximación más robusta como el algoritmo iterativo del punto más cercano (*ICP, Iterative Closest Point).* [2]

El presente TFG supone la profundización en la implementación de algunas variantes del algoritmo ICP para localización sobre diferentes escenas del mundo real.

Se enmarca en el proyecto Navegación y Despliegue de Robots en Entornos Desafiantes, también conocido como ROBOCHALLENGE (DPI2016-76676-R), que realiza el grupo de investigación de Robótica, Percepción y Tiempo Real (ROPERT) del Instituto Universitario de Investigación en Ingeniería de Aragón (I3A).

ROBOCHALLENGE se basa en investigaciones experimentales y teóricas en ámbitos desafiantes mediante robots aéreos y terrestres. Estos entornos desafiantes, son aquellos donde, por diferentes causas, fallan o no funcionan correctamente técnicas robóticas existentes actualmente o en desarrollo.

En estos entornos no se dispone de localización GNSS y la falta de una iluminación adecuada puede deshabilitar el uso de técnicas basadas en visión.

Este trabajo de fin de grado se enmarca dentro del objetivo de localización de robots en estos entornos utilizando únicamente sensores de rango como las nubes de puntos obtenidas por una cámara RGBD que no necesita de iluminación externa.

Además, supone la continuación de un proyecto anterior, ([3]), del cual se ha mantenido parte de la estructura base.

Los ficheros de gestión de tiempos, control de mensajes provenientes de la cámara y el programa de recogida de muestras, *tfg_recorder*, se han heredado con mínimos cambios.

Dentro del apartado de procesamiento, se ha prescindido de aquellas partes relacionadas con el uso del IMU y se ha añadido todo el contenido nuevo relacionado con las variantes del ICP utilizadas, así como su configuración.

La adaptación de los paquetes a ROS se ha tenido que realizar desde cero, ya que no se contaba con la estructura de archivos nativa de este entorno virtual.

2. OBJETIVOS Y REQUISITOS

El objetivo principal es desarrollar una técnica de autolocalización 3D basada en un mapa previo utilizando una cámara RGBD. Se basará en el algoritmo ICP (Iterative Closest Point).

Posteriormente se estudiará la posibilidad de incorporar un IMU (Inertial Measurement Unit).

El material necesario para llevar a cabo este proyecto es una cámara RGBD modelo ASUS Xtion PRO LIVE y un IMU modelo PhidgetSpatial 3/3/3. El software deberá ser adaptado específicamente a estos modelos.

A su vez, la localización inicial del sistema y el mapa del entorno en estudio se darán por conocidos.

2.1. Análisis de objetivos

El proceso de localización comienza con la captura del primer fotograma con la cámara RGBD. Se hará sobre una sección de nuestro entorno en estudio y sobre el cuál se ha creado un mapa en formato nube de puntos. La captura de la cámara vendrá también en formato nube de puntos.

El algoritmo ICP intentará alinear este fotograma sobre el mapa a base de iterar sobre el error existente entre la distancia de los puntos de ambas nubes.

Fig. 1.1. Ejemplo algoritmo ICP

La problemática existente con este algoritmo es que necesita de una buena estimación inicial para poder realizar una buena alineación de ambas nubes de puntos. Esto lo conseguimos realizando una estimación lineal de la posición partiendo de la localización actual y anterior que nos da el ICP.

Otro problema asociado es que el algoritmo ICP, en determinadas circunstancias, no genera buenos resultados y por ello no entrega transformaciones aceptables. Será en este momento en el que habrá que acudir a alguna de las variantes que este algoritmo presenta.

Esta tarea de alineación de nubes de puntos se realizará en post-procesado, lo que conlleva a la captura previa de los datos.

Como se demostró en estudios anteriores ([3]), la gestión en tiempo real de la captura y almacenamiento de las nubes de puntos resulta una tarea muy pesada a nivel de tiempos debido al gran tamaño (a nivel de bytes) que ocupan, por ello, se hará uso de la herramienta Rosbag, contenida dentro del entorno virtual ROS (Robot Operating System).

Lo que nos permite Rosbag es simular la captura de datos a una frecuencia elevada sin estar capturándolos a esa frecuencia. Es decir, se grabará una secuencia sobre nuestro entorno de trabajo, y posteriormente se reproducirá a una velocidad mucho menor para así permitir la captura de todos los mensajes por nuestra aplicación.

Como la estampa temporal de los mensajes es la misma reproduciéndolos a una velocidad que a otra, la aplicación los tomará a la frecuencia a la que se grabaron.

También se hará uso de los drivers necesarios para la gestión de estos mensajes. Se importarán como nodos de ROS (openni2 en caso de la cámara).

Falta por mencionar el entorno sobre el que localizaremos la trayectoria de nuestra secuencia. Este mapa debe ser construido en formato nube de puntos, y dado que tenemos una cámara RGBD se usará la herramienta RGBDSlam, implementada como un paquete de ROS.

3. ORGANIZACIÓN DE LA MEMORIA

Este documento se divide en 8 capítulos:

- Capítulo 1. Introducción.

  Se trata del actual capítulo, que pone en contexto al lector.

- Capítulo 2. Descripción del hardware y el software.

  Se explican los elementos harware utilizados en el proyecto y aquellas herramientas software que también han ayudado al desarrollo del mismo.

El desarrollo del proyecto se divide principalmente en cuatro etapas: creación del mapa, generación de la secuencia de movimiento, el cálculo de la transformación inicial en coordenadas del mapa y el procesamiento de la secuencia. A continuación se describen cada una de estas etapas.

- Capítulo 3. Creación del mapa.

  En este capítulo se explica sobre qué entorno crear el mapa y cómo.

- Capítulo 4. Generación de la secuencia.

  Se pasa a describir el funcionamiento del programa *tfg_recorder*.

- Capítulo 5. Cálculo de la transformación inicial.

  Aquí se describe la metodología usada para situar las nubes de la secuencia sobre las coordenadas del mapa.

- Capítulo 6. Procesamiento de la secuencia.

  Donde se describe el funcionamiento del programa *tfg_processor.*

En los últimos capítulos se realizan pruebas y se analizan los resultados.

- Capítulo 7. Pruebas de funcionamiento.

  En este capítulo se presentan tres entornos sobre los que se han realizado distintas pruebas.

- Capítulo 8. Conclusiones.

  Donde se comentan los resultados y se proponen mejoras y vías de continuación.

Se incorpora también a modo de anexos el código completo de las aplicaciones.

CAPÍTULO 2

Descripción del hardware y el software

A continuación se van a presentar aquellos componentes hardware que se han utilizado en el desarrollo del proyecto al igual que una explicación sobre las herramientas software empleadas.

1. HARDWARE

Los componentes hardware utilizados son una cámara RGBD y un IMU (Inertial Measurement Unit o unidad de medición inercial).

1.1. Cámara RGBD

Una cámara RGBD es un tipo de sensor capaz de proporcionar información sobre el color y la profundidad para cada pixel de la imagen.

La información del color es captada por una cámara digital y suele estar codificada en el espacio de color RGB.

La información de profundidad puede obtenerse con dos tipos de tecnologías, activas o pasivas.

Se entiende por tecnologías pasivas aquellas en las que la cámara únicamente recibe luz. La información de profundidad la obtienen a partir de identificar puntos de correspondencia en dos o más imágenes. [4]

Por el contrario, las tecnologías activas son aquellas que permiten la recepción y la emisión de luz. Por lo que no necesitan de una fuente externa de luz para funcionar.

Obtienen la información de profundidad proyectando un patrón de luz estructurado, por medio de un proyector infrarrojo, detectado por un sensor de imagen infrarrojo. [4]

La problemática de estos últimos sensores es que si se exponen a la luz del sol se pueden cegar, por lo que no se recomienda su uso en estos casos.

En el desarrollo de este proyecto se ha usado el sensor ASUS Xtion PRO LIVE. Un sensor RGB y de profundidad de la compañía ASUSTeK Computer Inc.

Con una distancia de uso entre los 0.8 m y los 3.5 m es capaz de grabar a 30 fps (*frames* per second o fotogramas por segundo) con una resolución de 640x480 o a 60 fps con una resolución de 320x240. En este proyecto se ha usado la primera configuración.

Su campo de visión va hasta los 58º en horizontal, 45º en vertical y 70º en diagonal.

En la Fig. 2.1 se indican las diferentes partes que componen el sensor. Los micrófonos que se muestran no se han utilizado en este proyecto.

Fig. 2.1. Partes del sensor ASUS Xtion PRO LIVE. [5]

2. SOFTWARE

A continuación se explicarán las herramientas software utilizadas en el desarrollo del proyecto, siendo estas: C++, ROS (Robot Operating System) y PCL (Point Cloud Library).

2.1. C++

Todo el proyecto se ha programado utilizando este lenguaje, ya que permite el uso de estructuras complejas de datos y es orientado a objetos.

2.2. ROS (Robot Operating System)

ROS es un pseudo sistema operativo de código abierto. Es una colección de herramientas, librerías y convenciones que tratan de simplificar la tarea de crear rutinas complejas y robustas sobre una gran variedad de robots. [6]

En definitiva, es un framework que, ejecutado sobre un sistema Unix, permite crear aplicaciones para la robótica.

Las tres partes más importantes de ROS que usaremos en este proyecto son el *master* (maestro), los *nodes* (nodos) y los *topics* (tópicos).

El master es el proceso que se encarga de gestionar todo. Registra los nodos que se van a ejecutar, interconecta los mensajes publicados con sus correspondientes receptores, permite lanzar aplicaciones contenidas en ROS, etc. Se puede decir que es el núcleo de ROS.

Los nodos son los módulos ejecutables y los tópics son los canales de comunicación entre éstos, que se encargan de la transmisión de los mensajes.

Los tópics más importantes que se han usado han sido */camera/depth/points*, */camera/depth/image* y */camera/rgb/image_raw*, que a su vez están contenidos dentro del driver de la cámara RGBD, *openni2_camera* y no hubiera sido posible capturar mensajes de ellos sin él. Para el desarrollo de este proyecto no es necesaria la utilización de los canales RGB, pero el uso del último topic, en el que sí tenemos estos canales, tiene su origen en la posibilidad de visualizar lo que capta la cámara cuando se graban secuencias para el procesamiento.

La versión de ROS utilizada en este proyecto es ROS Indigo Igloo.

También se ha hecho uso de la API roscpp, que ha permitido la implementación y compilación del proyecto en C++.

2.3. PCL (Point Cloud Library)

La PCL o Point Cloud Library es un conjunto de librerías bajo la licencia BSN dedicadas al procesamiento de imágenes en 2D, 3D y nubes de puntos. Contiene los últimos algoritmos (state-of-art) para el filtrado, la estimación de características, el mapeo, la identificación y la segmentación de imágenes y nubes de puntos. [7]

En este proyecto se ha hecho uso de esta librería para trabajar con la estructura de datos más importante que aquí tratamos, las nubes de puntos. Cabe decir que ROS cuenta con un tipo de dato nativo para trabajar con nubes de puntos, pero carece del resto de librerías necesarias que permiten filtrar, redimensionar, transformar o aplicar cualquier tipo de operativa sobre ellas.

Por ello, la operativa interna se ha realizado con el tipo de dato proveniente de PCL y sólo se ha recurrido al nativo de ROS a la hora de la publicación de los mensajes y su visualización.

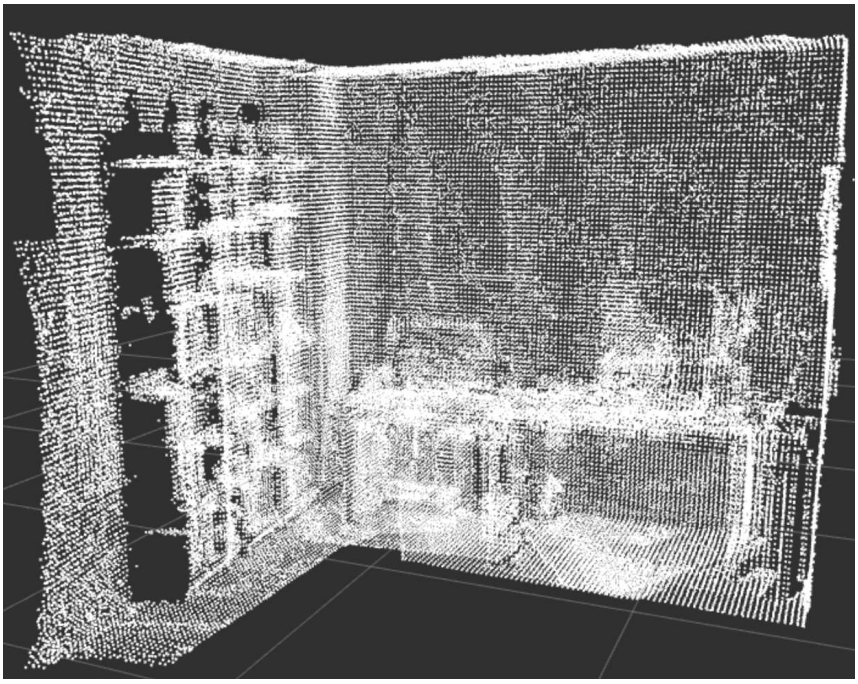


Fig. 2.2. Ejemplo nube de puntos.

El módulo más usado dentro de esta librería ha sido *Registration*, que almacena toda la configuración que se puede aplicar a los diferentes algoritmos de registro y una lista de estos algoritmos, entre los que destacan el punto a punto, el punto a plano, estimación 2D, estimación de Levenberg-Marquardt y la estimación de los mínimos cuadrados lineales (*LLS* por sus siglas en inglés) para el punto a plano. Además también contiene clases para establecer reglas de eliminación o aceptación de correspondencias y transformaciones en los algoritmos usados. Se trata de un módulo muy amplio que está en constante crecimiento.

2.4. RGBDSlam

RGBDSlam es un paquete de ROS que no viene preinstalado en el framework. Es una solución SLAM (Simultaneous localization and mapping) o mapeo y localización simultáneos que permite la creación de nubes de puntos registradas u octomaps. Cuenta con una interfaz visual aunque también puede usarse mediante comandos desde ROS. [8]

Con esta aplicación podemos crear mapas a partir de una cámara RGBD y la composición de las nubes de puntos que va capturando. Se basa en la utilización de puntos de interés y los descriptores SIFT, SURF y ORB para relacionar los fotogramas. [9]

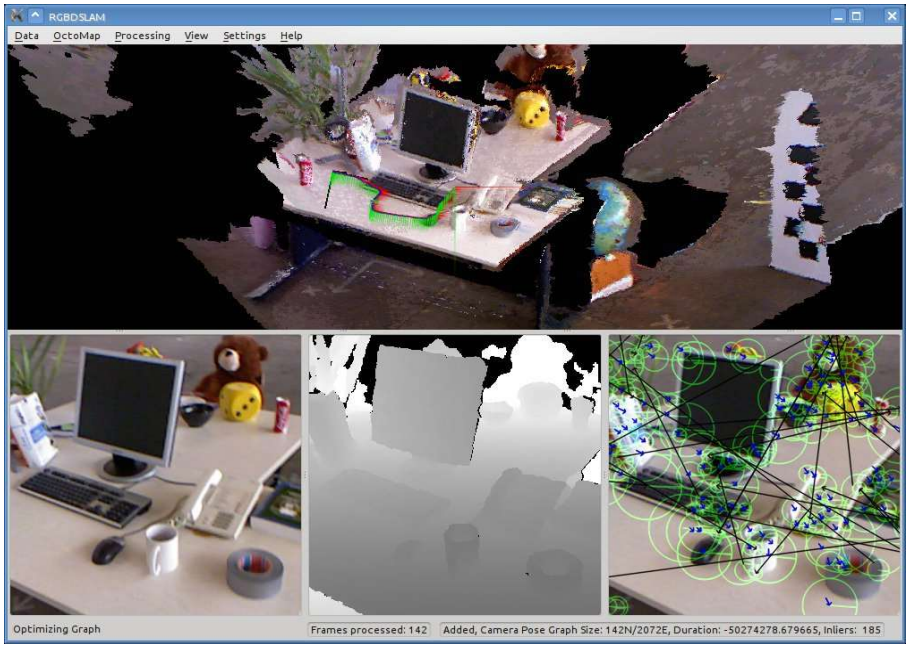


Fig. 2.3. Interfaz de la aplicación RGBDSlam. [10]

2.5. RViz

Es un paquete que por defecto viene incluido con la instalación de ROS, por lo tanto funciona sobre éste, y cuenta con una interfaz visual que permite la visualización y configuración de los mensajes que se transmiten.

A continuación podemos ver la interfaz de esta aplicación.

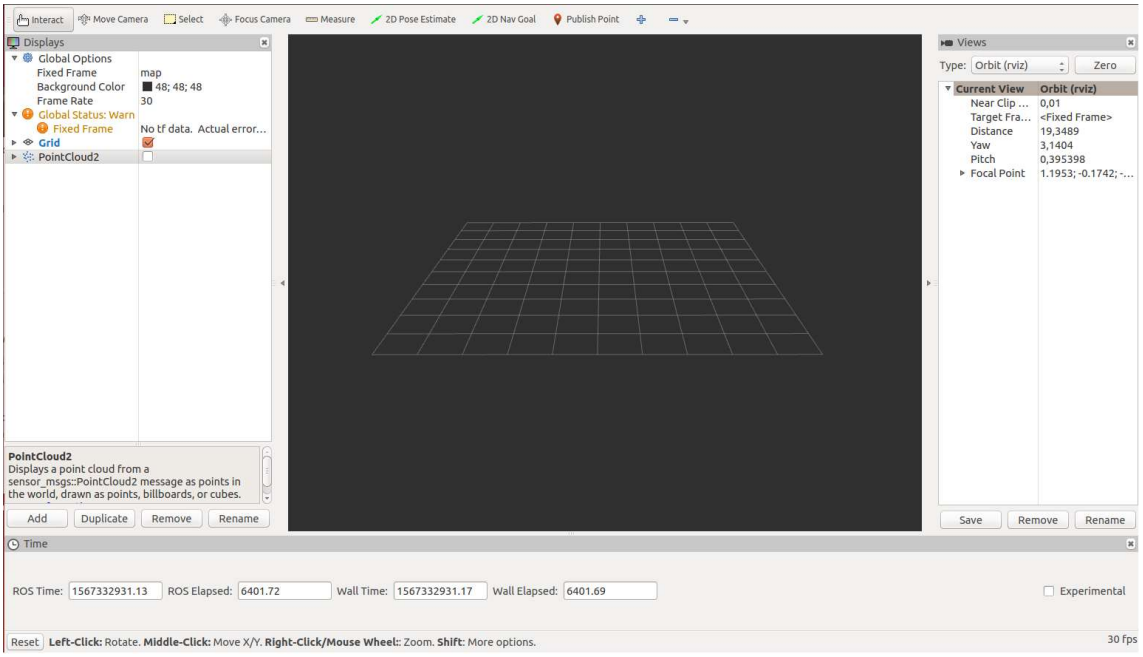


Fig. 2.4. Interfaz del programa RViz.

CAPÍTULO 3

Creación del mapa

Como ya se ha comentado en el primer capítulo, para poder localizar nuestra cámara necesitamos previamente disponer del entorno sobre el que vamos a hacerlo.

Primero es necesario precisar sobre qué entorno crear el mapa y después cómo crearlo.

1. ELECCIÓN DEL ENTORNO

La limitación más importante en este apartado es el material del que se dispone. Al contar únicamente con una cámara RGBD es necesario atender a sus características para así poder escoger un entorno lo más óptimo posible.

Según las características de nuestra cámara, tenemos unos rangos de distancia entre los 0.8 m y los 3.5 m. Tampoco puede incidir la radiación solar sobre el sensor de la cámara porque podría cegarlo.

Existe otra limitación, y son aquellas zonas planas, sin ningún tipo de rugosidad u objetos sobre ellas, que dan lugar a la incertidumbre de saber si corresponden a una zona del mapa o a otra.

Por esto, interesaría que fuera un entorno en el que exista una gran variedad de detalles.

Un entorno idóneo sería el interior de cuevas, grutas o pozos, grabando dentro de las distancias recomendadas por la cámara. Y al ser entornos oscuros y protegidos de la radiación solar tampoco correríamos el peligro de cegar el sensor.

Otro entorno sobre el que se puede testear es en el interior de una vivienda. Las distancias pueden ser adecuadas y existe la posibilidad de no dejar entrar la luz exterior.

Por simplificación se ha escogido la segunda opción, crear el mapa sobre la habitación de una vivienda. Se ha atendido a la necesidad de que exista variedad de detalles.

2. CONSTRUCCIÓN DEL MAPA

De acuerdo a lo anteriormente comentado respecto a las limitaciones del material disponible para este proyecto, la creación del mapa la podemos realizar únicamente con la cámara RGBD. Esto nos deja pocas opciones, que son RTAB-Map y RGBDSlam v2.

RTAB-Map basa su método de localización en una bolsa de palabras (*bag-of-words*) que crece de manera incremental y no mediante entrenamiento previo. Esta bolsa está compuesta por descriptores SURF (*Speeded-Up Robust Features*), y sólo aquellos con detalles prominentes son extraídos de una imagen con OpenCV.

Debido a la gran la gran dimensión a la que se puede llegar con la extracción de descriptores, RTAB-Map utiliza una estructura arbolada con 4 árboles kd, que no son más que una estructura de datos que organiza los puntos en un espacio euclídeo de k dimensiones. En este caso, los puntos de ese espacio euclídeo se corresponderían con los descriptores SURF.

Además de la utilización de árboles kd para mejorar la búsqueda de descriptores, RTAB-Map se basa en un sistema del gestionado de la memoria que diferencia entre memoria de trabajo y memoria de largo plazo. De esta manera almacena las localizaciones más observadas y recientes en la memoria de trabajo, y las demás las transfiere a la memoria de largo plazo.

Además de todo esto, RTAB-Map utiliza un método de cerrado de bucles que determina, gracias a los descriptores SURF, si una localización coincide con otra ya tomada. Se apoya en un filtro bayesiano para estimar la probabilidad de que la localización actual se corresponde con otra ya tomada y almacenada en la memoria de trabajo. [11]

RGBDSlam v2 tiene un funcionamiento similar en algunos aspectos. Primero se realiza una extracción de descriptores, que pueden ser SIFT, SURF u ORB, para luego asociarlos a puntos 3D y con esto poder construir una gráfica de posición que es optimizada con el framework g^2o, que se encarga de minimizar errores en las funciones gráficas no lineales. [12] [9]

El cerrado de bucles consiste en un procedimiento que explota el conocimiento sobre el bucle prefiriendo aquellos *frames* candidatos que están más cerca una vez se ha encontrado una coincidencia. Utiliza una búsqueda basada en el vecindario geodésico y el cálculo de un árbol recubridor mínimo para lograr establecer estas coincidencias. [9]

En conclusión podemos decir que ambos programas tienen sus puntos fuertes y sus puntos débiles. RTAB-Map tiene un método de cerrado de bucles mucho más completo pero también más pesado a nivel de consumo de recursos, sin embargo RGBDSlam posee una extracción de descriptores mucho más polivalente y el cerrado de bucles es más ligero aunque no tan robusto.

Dado que el equipo sobre el que se ha desarrollado este proyecto ya traía instalado el paquete de RGBDSlam y también por su polivalencia en el establecimiento de descriptores se ha decido continuar con el uso de éste.

La creación del mapa con RGBDSlam v2 es relativamente sencilla una vez se tiene todo instalado.

La ejecución del programa se hace desde el entorno virtual ROS. Y toda la configuración de parámetros se recoge en un archivo con extensión *.launch*, que son algo parecido a los ejecutables de ROS.

Dentro de la interfaz del programa se pueden distinguir las imágenes tomadas a color (1), la imagen de profundidad (2), la imagen con toda la visualización de descriptores (3) y el mapa a color (4).

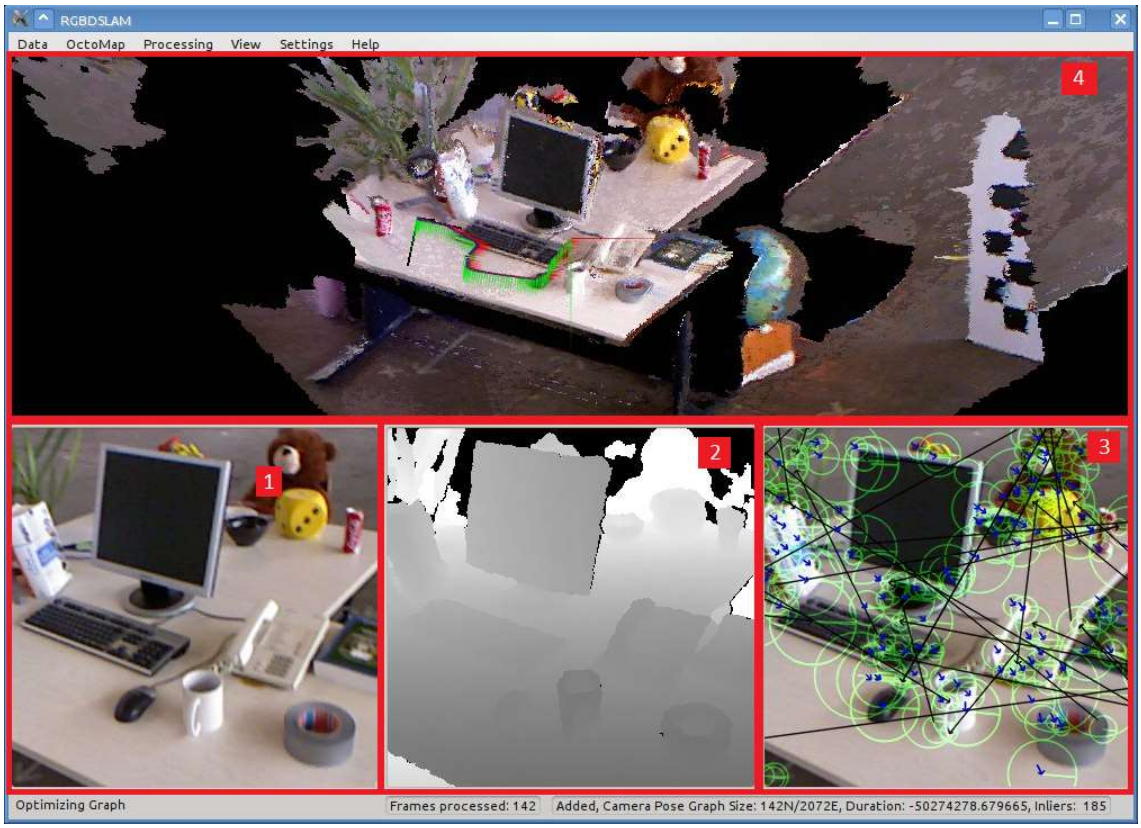


Fig. 3.1. Partes de la interfaz de RGBDSlam. [10]

Como se ha comentado anteriormente, el método de cerrado de bucles es menos robusto que el de RTAB-Map, aunque frente a desplazamientos lineales con poca separación entre capturas responde relativamente bien. En las siguientes imágenes se puede observar cómo ha fallado el algoritmo de cerrado de bucles cuando se gira la cámara y se desplaza a la vez.

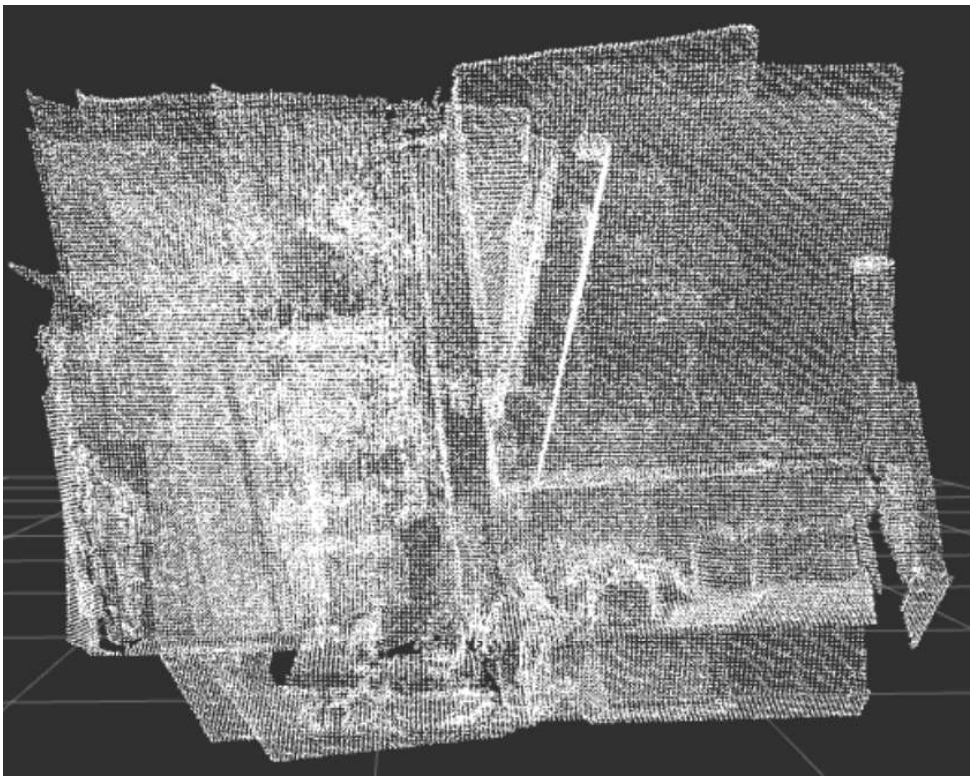


Fig. 3.2. Mapa mal formado debido a una mala transformación entre *frames*.

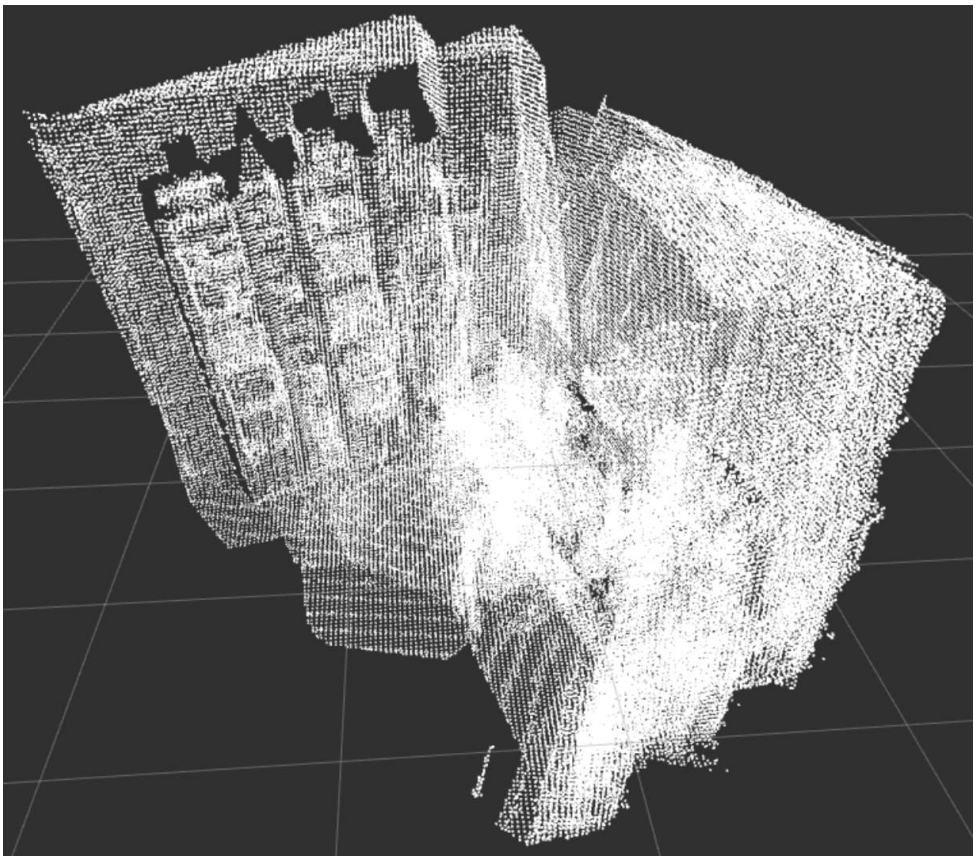


Fig. 3.3. Mapa mal formado debido a giros y movimientos lineales al mismo tiempo.

Después de haber realizado pruebas con varios entornos y varias veces con cada uno se ha observado que la extracción de detalles con descriptores ORB es la que mejor ha creado los mapas. También se han obtenido mejores resultados optimizando únicamente todos los *frames* desde el último encontrado hasta el actual y utilizando como método de optimizacón el llamado *pcg*, que se basa en una buena estimación inicial de la posición que inicia el mapa.

Otros parámetros importantes han sido los que conforman el algoritmo de cerrado de bucles, comparaciones con capturas aleatorias, comparaciones con *frames* secuenciales y comparaciones con el vecindario geodésico. Fijando valores altos, como podrían ser 6, 8 o 10, se observa mayor desubicación en la formación del mapa. Los valores óptimos se han encontrado en torno a 4 para los tres parámetros.

Atendiendo a esta parametrización se han conseguido crear varios mapas correctamente.

Otro aspecto importante a la hora de crear el mapa con RGBDSlam v2 es el método de toma de *frames* (o capturas) del entorno. Puede ser mediante una grabación con rosbag, una grabación en tiempo real y toma de *frames* uno a uno. Los mejores resultados se han obtenido tomando cada *frame* por separado sin dejar demasiada distancia lineal entre ellos.

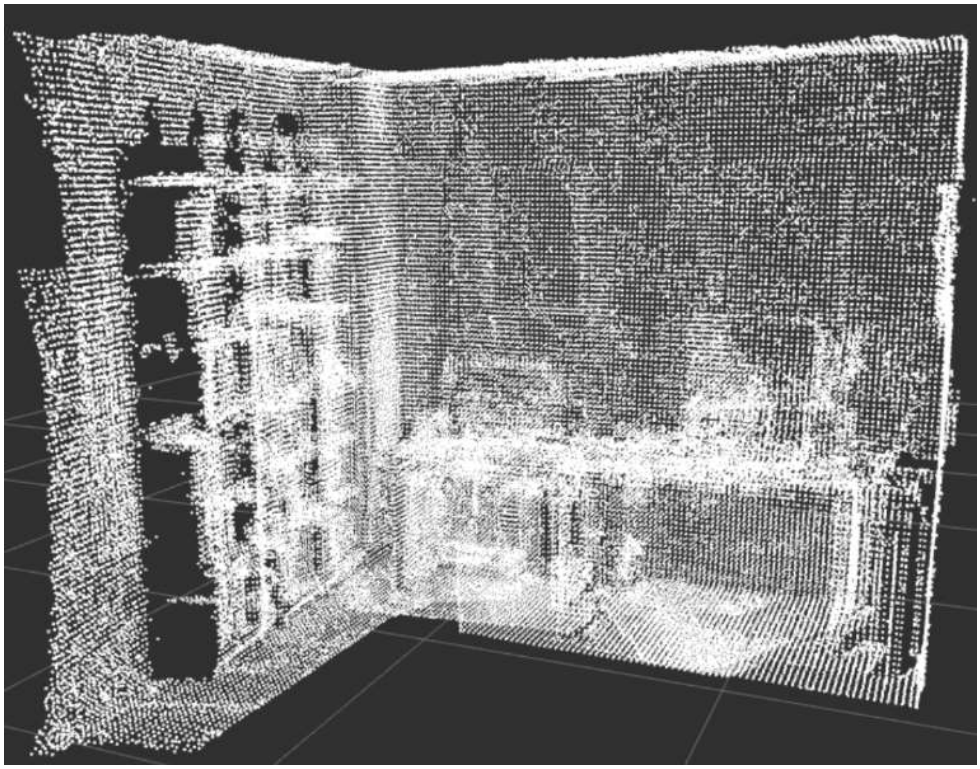


Fig. 3.4. Mapa bien conformado siguiendo la parametrización descrita.

Otro ejemplo de mapa creado con un resultado aceptable es el siguiente:

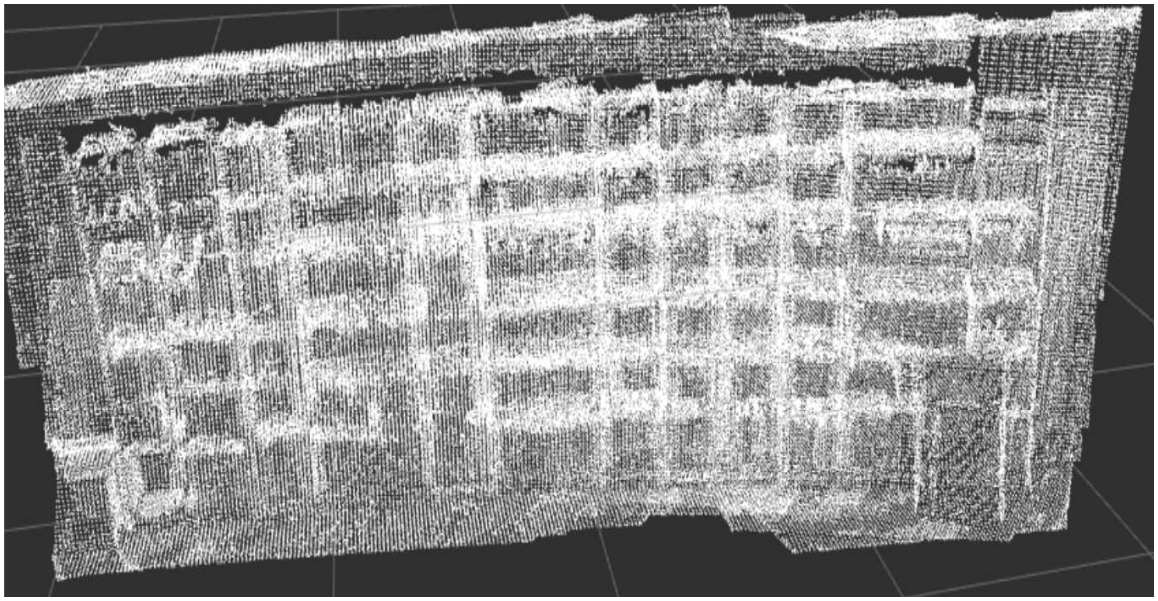


Fig. 3.5. Mapa creado con correcta parametrización.

CAPÍTULO 4

Generación de la secuencia de movimiento

Una vez que ya se dispone del entorno se procede a generar la secuencia que se tratará de localizar sobre ese entorno. Este proceso de localización se hará offline, es decir, en postprocesado, de manera que en este capítulo se detalla el grabado y guardado de manera óptima de los datos para un posterior análisis.

1. GRABACIÓN PREVIA CON ROSBAG

Como ya se adelantó al comienzo de este proyecto, se va a utilizar la herramienta *rosbag*, contendia en el entorno virtual ROS.

Esta herramienta permite el grabado de cualquier topic que esté publicando y también admite diversas configuraciones como compresión de la grabación, imprimir la información de una grabación, convertir grabaciones usando Python, comprobar el estado una grabación, reproducir grabaciones, etc. [13]

En este caso, se aprovechará la función de grabar y reproducir. Se grabarán los topics necesarios para capturar toda la información del movimiento descrito por la cámara en el entorno de estudio. Posteriormente se reproducirán a una velocidad menor que permitirá capturar todos los mensajes a la aplicación encargada de tratarlos y transformarlos.

Cabe decir que esta herramienta no sería capaz de capturar ningún mensaje enviado por la cámara si no fuera por el driver *openni2*. Este driver viene incluido con la instalación de ROS y lo podemos lanzar ejecutando el archivo *openni2.launch*.

Los topics utilizados en esta fase son los siguientes:

1. */camera/depth/points*
2. */camera/rgb/image_raw*
3. */camera/depth/image*
4. */camera/depth_registered/camera_info*
5. */rosout*
6. */tf*

Algunos de estos topics eran necesarios para posteriores visualizaciones con RViz, y otro, */camera/rgb/image_raw*, se almacenó para poder comprobar a posteriori el resultado de la grabación y descartarla si fuese necesario.

Se ha buscado grabar con movimientos suaves y lentos para poder capturar *frames* no muy separados entre sí espacialmente y facilitar la localización. Esta secuencia generará un archivo *.bag*.

2. ESTRUCTURA DE DATOS DE LA APLICACIÓN

Esta aplicación es la que se encarga de leer y transformar la información proveniente del archivo *.bag* generado anteriormente. Para su posterior procesamiento es necesario capturar la estampa temporal de los mensajes, pero no es necesario que sea una estampa absoluta, si no relativa al comienzo de captura de datos.

Por ello se creará una escala de tiempos específica a este proyecto. En el siguiente esquema se ejemplifica la captura de los mensajes:

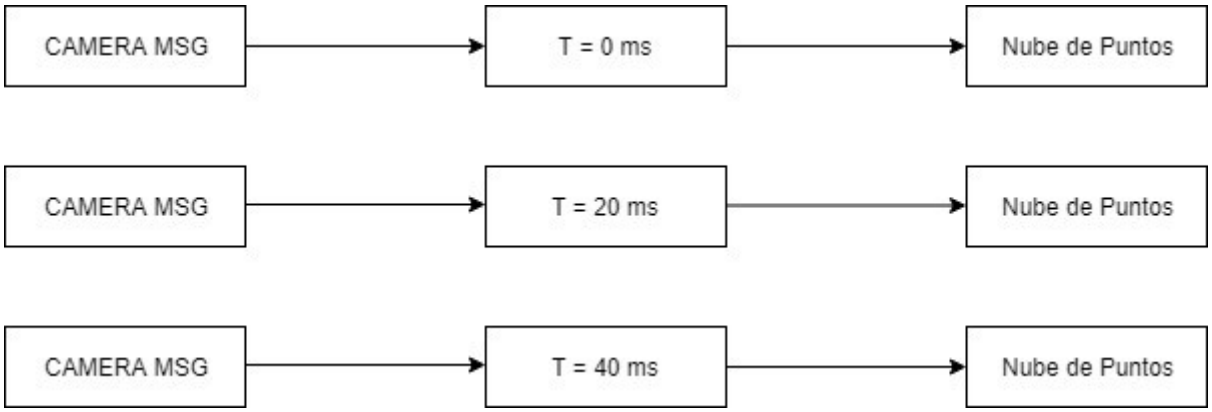


Fig. 4.1. Ejemplo de secuencia.

Esta secuencia se va a almacenar en ficheros. PCL tiene su formato para las nubes de puntos, *.pcd* (Point Cloud Data), y será este formato el utilizado para almacenar todas las nubes de puntos. Para el resto de información se utilizarán ficheros de texto, *.txt*.

Esta estructura de ficheros tiene que dar cabida a todas las nubes de puntos que se capturen de la secuencia, el número de las mismas y la estampa temporal de cada una.

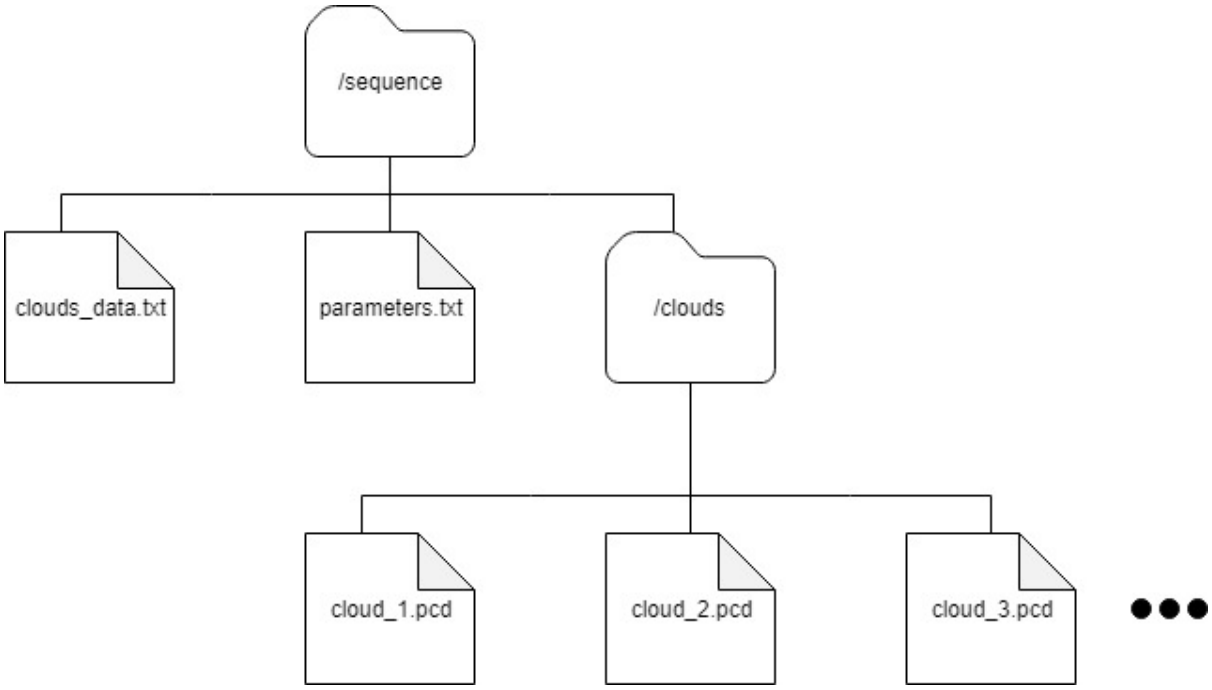


Fig. 4.2. Sistema de ficheros utilizado en la aplicación de grabado de la secuencia

Dentro de la carpeta */sequence* podremos encontrar lo siguiente:

7. ***clouds_data.txt***: Almacena el nombre de cada nube de puntos, de acuerdo a un formato fijo, y el tiempo en milisegundos desde el inicio de la secuencia.

8. ***parameters.txt***: Almacena el número total de nubes de puntos, descartando siempre la última para evitar malas escrituras por cierres inesperados de la aplicación.

9. ***/clouds***: En esta carpeta se almacenan las nubes de puntos extraídas.

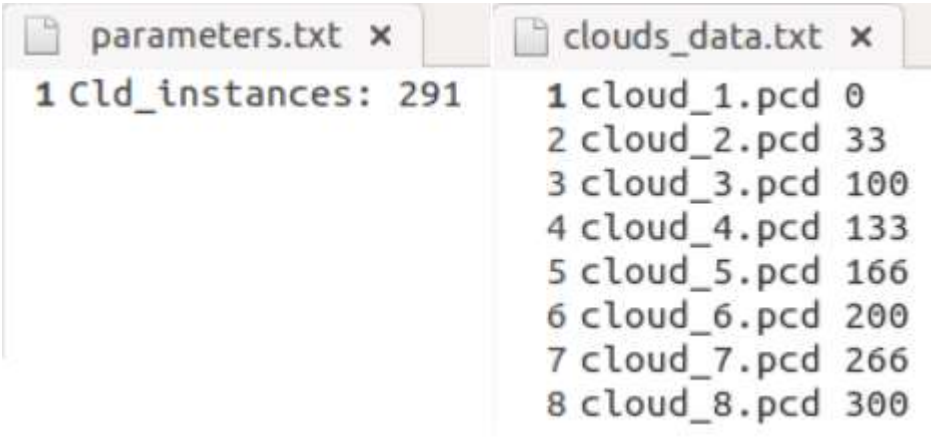


Fig. 4.3. Ejemplo de *parameters.txt* y *clouds_data.txt*.

3. DESCRIPCIÓN DE LA APLICACIÓN

Esta aplicación de grabado se implementará como un nodo de ROS, por lo que necesita de la ejecución de roscore (el master), y se llamará *tfg_recorder*.

Al crear la carpeta de este nodo, automáticamente se generarán dos carpetas dentro de ella, *src* e *include*, y dos archivos, *CMakeLists.txt* y *package.xml*, que albergan toda la configuración de nuestro nodo. Será dentro de la carpeta *tfg_recorder* donde introduzcamos la carpeta */sequence* anteriormente mencionada.

El programa comienza con la inicialización de nuestro nodo *tfg_recorder* y subscribiéndose al topic */camera/depth/points*.

El cierre del programa, señal SIGINT, se ha modificado para asegurar la integridad de los ficheros y evitar fallos de escritura y cierre.

Finalmente se llama al spinner, que es la manera que hay en ROS con la API para C++ de crear un bucle que acepte separación en varios hilos de ejecución e interrupciones del sistema.

No existe la necesidad de crear una función *main* ya que lo único que vamos a gestionar son las interrupciones de la llegada de una nube de puntos, y eso es posible realizarlo con el spinner y una función *callback* que se ejecute cuando se recibe una nube de puntos.

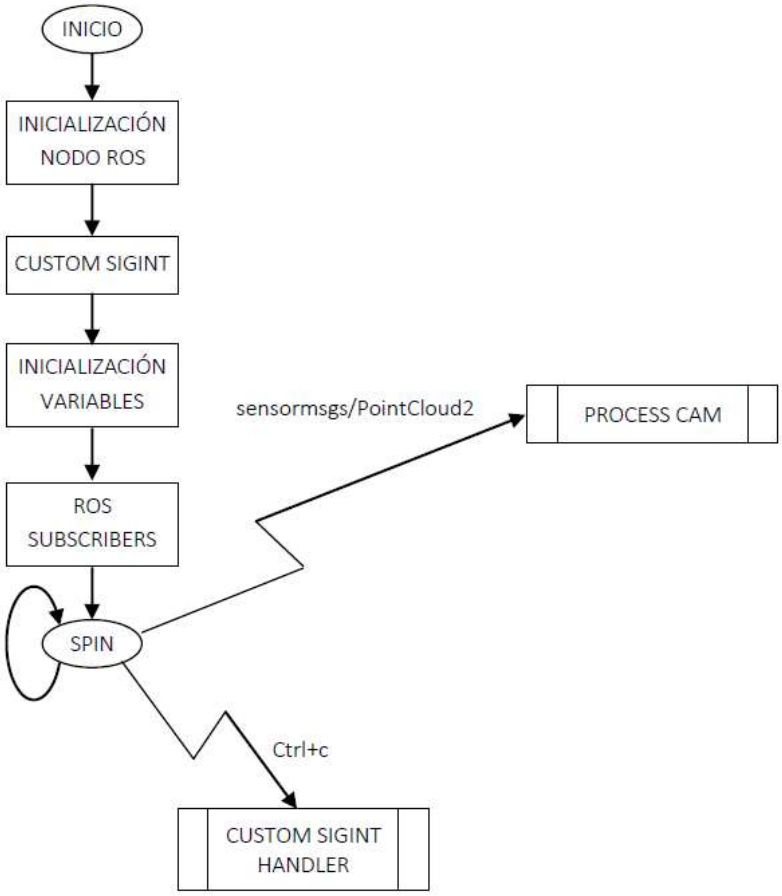


Fig. 4.4. Diagrama de flujo de la aplicación *tfg_recorder*.

4. LECTURA DE LA CÁMARA

La función encargada de recoger los datos de la secuencia almacenada como archivo *.bag* se llama processCam, y se ejecuta cada vez que llega un mensaje de tipo *sensor_msgs::PointCloud2*. Este formato es el nativo de ROS para tratar las nubes de puntos, pero no deja mucho margen a la operativa, así que es necesario transformar al formato de PCL, *.pcd*, y guardar el archivo.

Finalmente se calculan los tiempos para conocer la frecuencia de guardado.

A continuación se presenta el diagrama de flujo de esta función. Se utiliza un mutex para evitar el acceso simultáneo a variables compartidas.

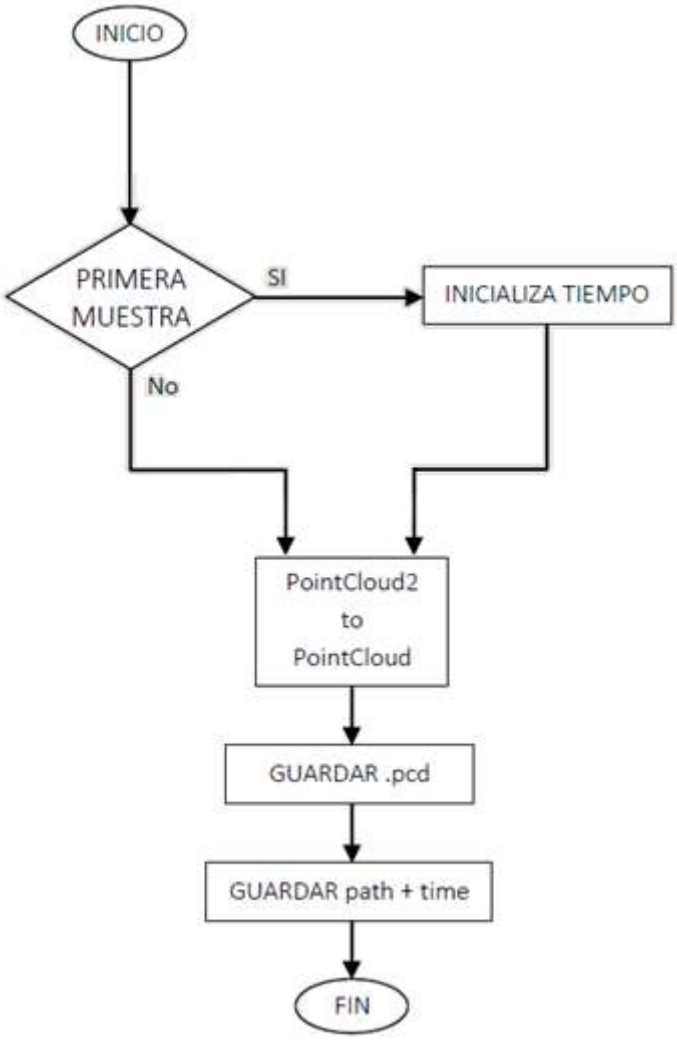


Fig. 4.5. Diagrama de flujo de la función que lee los mensajes almacenados provenientes de la cámara.

CAPÍTULO 5

Cálculo de la transformación inicial

Una vez tenemos el mapa del entorno creado y la secuencia a tratar, necesitamos situar sobre un mismo sistema de referencia ambas escenas. Como estamos trabajando sobre un mapa y el objetivo es localizar una secuencia dentro de ese mapa, será este el sistema que consideremos como fijo.

Para implementar esto es necesario hallar la transformación que posiciona la primera nube de puntos de la secuencia sobre su localización en el mapa. Esto se tendrá que realizar de manera visual, ajustando de manera manual hasta colocar la primera nube y después, de manera opcional aplicar el algoritmo ICP para que nos entregue de manera precisa esa transformación.

1. DESCRIPCIÓN DE LA APLICACIÓN

El funcionamiento de la aplicación es sencillo. Inicializa el nodo de ros llamado *tfg_set_origin*, inicializa las variables necesarias, carga la información del mapa y la primera nube de puntos, le aplica las transformaciones que se van estimando de manera visual y publica las nubes para poder visualizar los cambios en RViz.

Hay que tener en cuenta que cada vez que es necesario hacer transformaciones a las nubes, se debe parar la ejecución del programa, modificar, compilar y volver a ejecutar con los cambios.

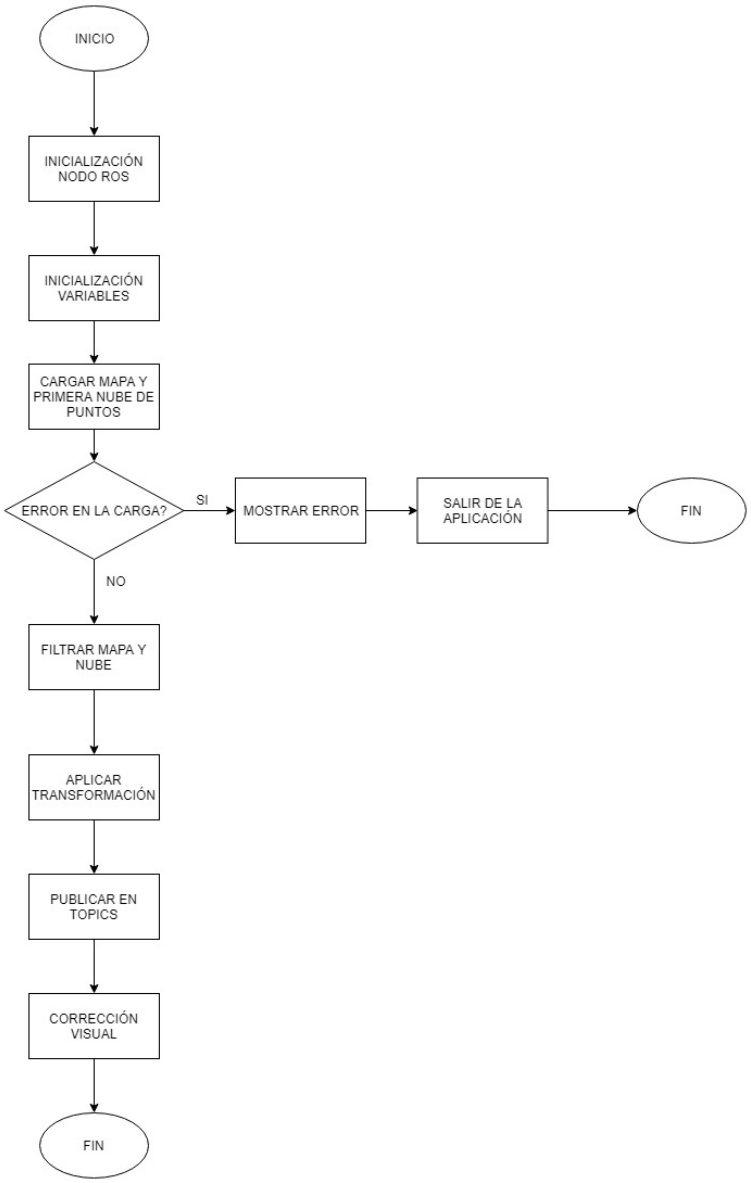


Fig. 5.1. Diagrama de flujo de la aplicación *tfg_set_origin*.

Una vez se tiene acotada la posición de la primera nube de puntos en el mapa, se puede escoger esa transformación y utilizarla como origen para la aplicación de procesado o se puede aplicar el ICP para obtener la posición precisa. La irrelevancia de esto último se debe a que en la primera iteración del algoritmo de localización ya se aplica el ICP, por lo tanto no importa mucho aplicarlo antes o después.

CAPÍTULO 6

Procesamiento de la secuencia de movimiento

En este capítulo se va a explicar el funcionamiento del programa *tfg_processor*, que utiliza los datos de la secuencia previamente grabada y los procesa para lograr estimar la trayectoria que se ha seguido en dicha secuencia.

1. ESTRUCTURA DE DATOS

Los datos recogidos con *tfg_recorder* y el mapa creado con RGBDSlam constituyen los parámetros de entrada de nuestro programa.

La estructura de ficheros creada comienza en el directorio */map*, y es la ubicación de esta carpeta la que toma como argumento *tfg_processor* para así ir cargando y procesando los datos.

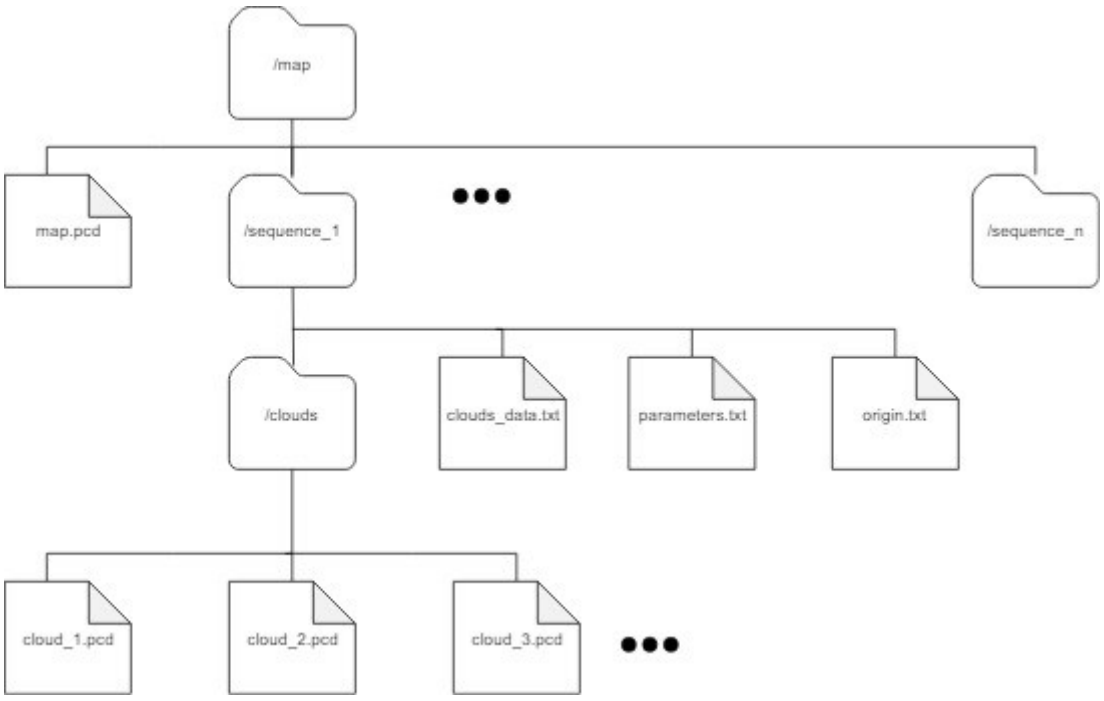


Fig. 6.1. Sistema de ficheros de *tfg_processor*

10. El archivo *map.pcd* contiene el mapa creado con RGBDSlam en el formato nativo de PCL (*.pcd*).
11. Se pueden almacenar tantas secuencias como disponibilidad de espacio en el disco se tenga, simplemente hay que atender a la nomenclatura, que es *sequence_<n>*, con *<n>* el número de la secuencia. Este argumento puede ser especificado previamente a la ejecución del programa.
12. Dentro de cada carpeta de cada secuencia se tiene la carpeta */clouds*, que se corresponde con las nubes de puntos generadas con *tfg_recorder*, y los archivos *clouds_data.txt*, *parameters.txt*, *origin.txt.*
    - *clouds_data.txt* contiene en cada línea el nombre de la nube de puntos y la estampa temporal. Este archivo es generado por *tfg_recorder*.
    - *parameters.txt* recoge el número total de nubes de puntos que hay.
    - *origin.txt* almacena el origen de la secuencia en coordenadas del mapa en forma de matriz.

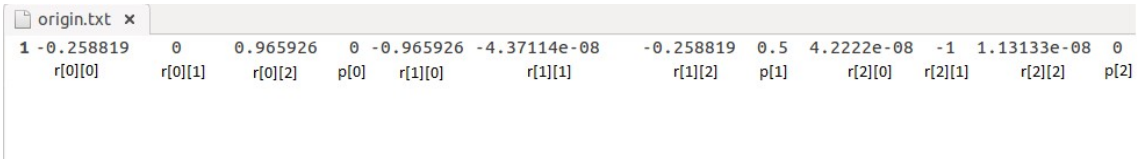


Fig. 6.2. Ejemplo de *origin.txt*, posiciones en matriz de rotación, r, y vector posición, p.

2. DESCRIPCIÓN DE LA APLICACIÓN

El programa encargado de procesar toda la secuencia se llama *tfg_processor*, está implementado como un nodo ROS, por lo que necesita de la ejecución del máster, *roscore*, para poder ser ejecutado.

La interrupción del sistema ha sido modificada para cerrar los ficheros adecuadamente y asegurar así su integridad.

A continuación se presenta el diagrama de flujo de esta aplicación:

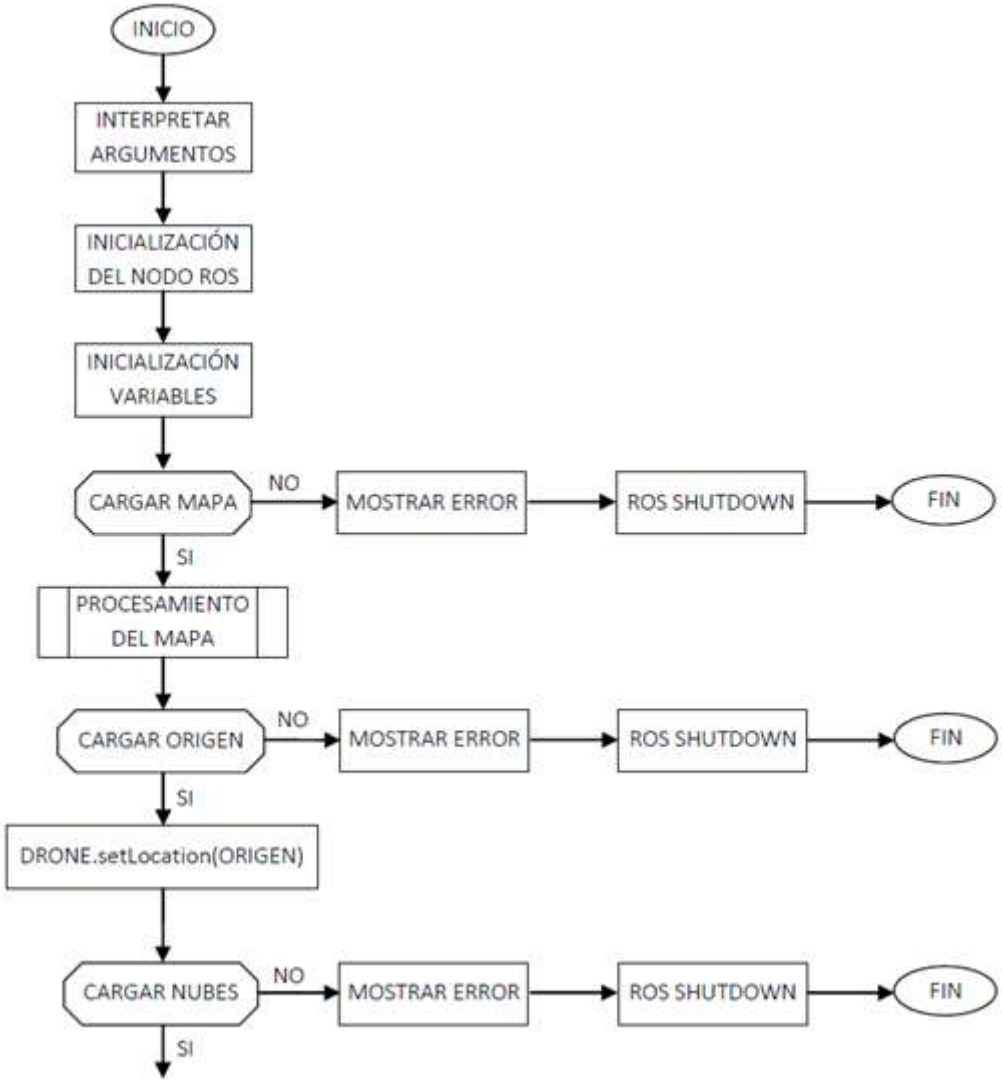


Fig. 6.3.a Diagrama de flujo de *tfg_processor*.

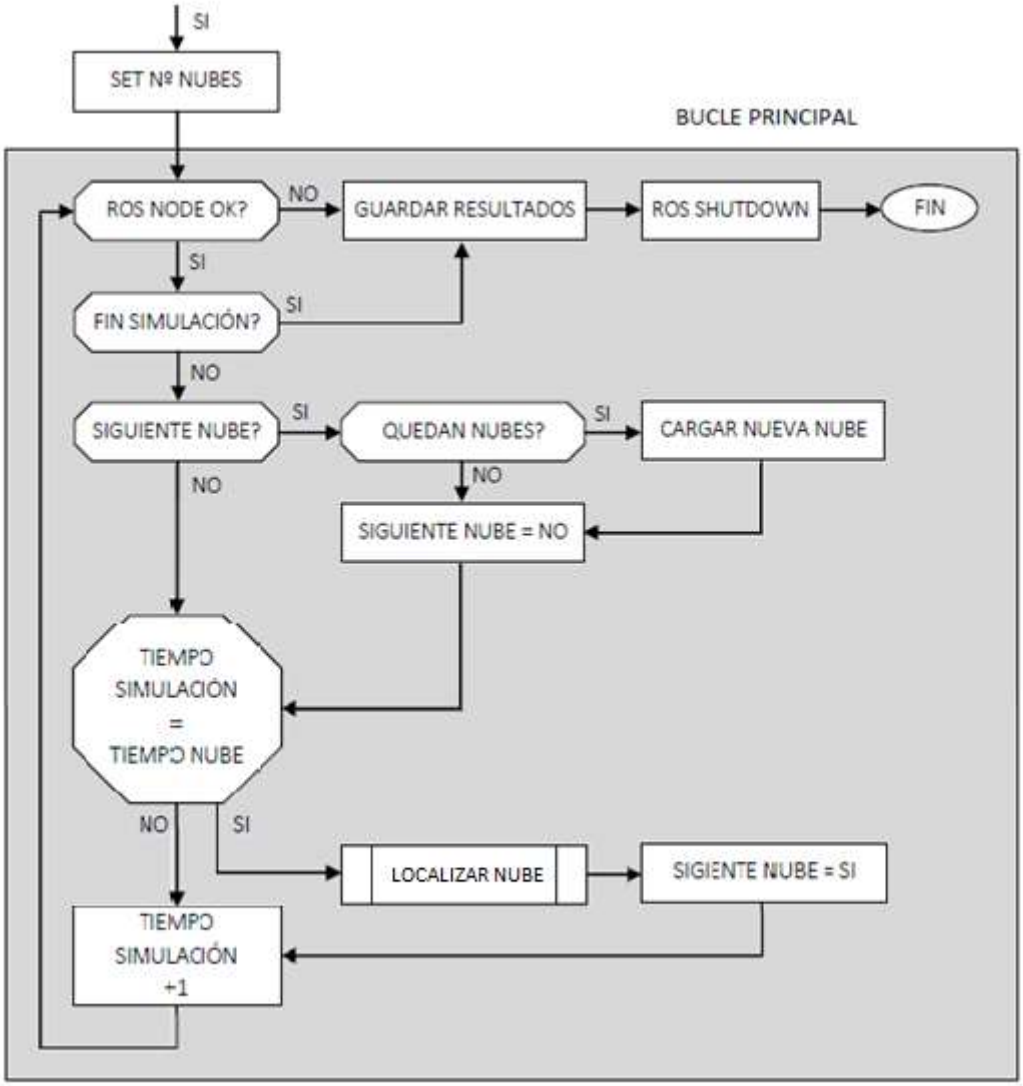


Fig. 6.3.b Diagrama de flujo de *tfg_processor*.

El programa trata de emular una aplicación de tiempo real y trabaja como un simulador. Comienza con la carga de los ficheros, y una vez cargados e inicializado todo el sistema, comienza con a procesar la secuencia. En el momento en que se detecta un error en el cargado de los datos o ficheros se cancela la ejecución mostrando un mensaje de error.

Se tiene un reloj con precisión de un milisegundo para gestionar internamente la aplicación. Cada iteración se incrementa una unidad y si coincide con la estampa temporal de la entrada, en este caso nube de puntos, la procesa. Así se consigue el procesado de la secuencia en el orden en el que fue grabada.

Se impide la carga de nuevas nubes de puntos mientras se está llegando a la estampa temporal de la última cargada mediante una variable. De esta manera se evita sobreescribir la nube que se está procesando en ese momento.

El procesado finaliza cuando no quedan nubes de puntos que procesar. Es en ese momento en el que se vuelcan y guardan los resultados.

3. PROCESAMIENTO DEL MAPA

El mapa no es más que una composición de muchas otras nubes de puntos (creado con RGBDSlam) y por ello contiene una alta densidad de puntos.

El hecho de que sea una composición de nubes hace que aparezcan ciertos problemas y efectos indeseables que son necesarios filtrar para minimizarlos en la medida de lo posible.

Dependiendo de la calidad del mapa y de las nubes de puntos, en el procesamiento de algunas secuencias y entornos se ha hecho necesaria la aplicación de los siguientes filtros:

13. **Voxel Grid Filter**: Este filtro divide el espacio en cubos 3D de un tamaño, el cual se corresponde con la resolución de la nube de puntos que se desea obtener. Calcula el centroide de los puntos contenidos en cada cubo, es decir, la media de las coordenadas de estos puntos. Se obtiene una nube discretizada en cubos en los que se encuentra un único punto por cubo. Así se consigue una reducción en la resolución y un filtrado paso bajo de la nube.

14. **Eliminación de *Outliers***: Los *outliers* son puntos numéricamente distantes del resto de una muestra. Su origen puede ser debido a una mala captura del sensor o a la existencia de perfiles con gradientes de profundidad muy grande. Para eliminarlos se aplican dos métodos distintos: *Statistical Outlier Removal Filter* y *Radius Outlier Removal Filter*.

    - *Statistical Outlier Removal Filter*: Este algoritmo itera dos veces en la nube de puntos. En la primera hace una clasificación de los puntos basada en el cálculo de la media y desviación estándar de las distancias de un punto a sus k-vecinos y en base a esto en la segunda iteración separa los puntos en *inliers* u *outliers*.
    - *Radius Outlier Removal Filter*: Este algoritmo sólo itera una vez, y clasifica los puntos en *outliers* dependiendo de si tiene más vecinos a un número dado, configurado como parámetro de entrada, o menos, el punto es considerado como bueno o se descarta, respectivamente.

4. LOCALIZAR NUBE DE PUNTOS

Después de haber cargado una nube de puntos y haber llegado al tiempo de simulación que coincide con la estampa temporal de la nube, entramos a procesar la información que ha capturado la cámara.

Mediante la aplicación del algoritmo ICP se intenta conseguir una transformación entre la posición anteriormente estimada y la actual, pero, ¿En qué consiste el algoritmo ICP y cómo consigue localizar la nueva nube de puntos? A continuación se da una explicación detallada a estas preguntas.

4.1. Funcionamiento del algoritmo ICP

El ICP es un algorimo iterativo que busca minimizar la suma del error cuadrático entre un punto sometido a una transformación, perteneciente a los puntos objetivo (*target*), y su correspondiente en el conjunto de los puntos de origen (*source*). Esto se puede formalizar con la siguiente expresión:

$$E(R,t) = 1/N_p \sum_{i=1}^{N_p} \left\| x_i - Rp_i - t \right\|^2 \qquad (6.4.1)\ [14]$$

Si las correspondencias entre puntos son conocidas, las rotaciones y traslaciones relativas se pueden calcular, si no, es necesario introducir algunas modificaciones.

Si las correspondencias entre puntos son conocidas, las rotaciones y traslaciones relativas se pueden calcular sin iterar, si no, es necesario introducir algunas modificaciones.

Para resolver el problema de no conocer las correspondencias entre puntos es necesario hacer el cálculo iterativo de la matriz de rotación y el vector de traslación hasta encontrar un mínimo local en la función del error, es decir, la función converge. Si los puntos están inicialmente muy distantes el algoritmo falla y no converge, de ahí la necesidad de estimar adecuadamente la posición inicial que toma el algoritmo en cada iteración.

La implementación de este algoritmo puede encontrarse dentro de la librería PCL, y dentro de esta existen muchas clases para la estimación de correspondencias, variantes del algoritmo ICP, métodos de descarte de malas aproximaciones, etc.

Por defecto, la implementación que hay en la librería PCL del algoritmo ICP aplica una variante basada en el *Singular Value Decomposition*, que consiste en substraer los centros de masa de cada punto tanto en el *target* (subconjunto de puntos *X*) *como* en el *source* (subconjunto de puntos *P*) y recalcular el error (los puntos se escogen recorriendo punto a punto la nube del *target* y sobre cada uno de éstos elegir el primer vecino encontrado):

$$\mu_x = \frac{1}{N_x} \sum_{i=1}^{N_x} x_i \quad Centro\ de\ masa\ de\ X$$
$$\mu_p = \frac{1}{N_p} \sum_{i=1}^{N_p} p_i \quad Centro\ de\ masa\ de\ P \qquad (6.4.2)\ [14]$$

Aplicando el *SVD* a los nuevos conjuntos de puntos y extrayendo los valores singulares de la aplicación de dicho algoritmo, obtenermos la siguiente fórmula de cálculo del error:

$$E(R,t) = \sum_{i=1}^{N_p} (\left\| x_i' \right\|^2 + \left\| y_i' \right\|^2) - 2(\sigma_1 + \sigma_2 + \sigma_3) \qquad (6.4.3)\ [14]$$

Con σ_1, σ_2 y σ_3 los valores singulares

En este proyecto se usan alternativamente dos implementaciones del ICP. Una es la ya mencionada, basada en *SVD*, conocida también como punto a punto, y la otra es la conocida como punto a plano.

El punto a plano se basa en el cálculo del mínimo que resulta de la suma de la distancia cuadráticaentre cada punto del *source* y el plano tangente en el *target*. [15]

$$M_{opt} = \arg\min_M \sum_I ((M \cdot s_i - d_i) \bullet n_i)^2 \qquad (6.4.4)\ [15]$$

Donde M_{opt} y M son matrices 4x4 de transformaciones de cuerpo rígido 3D, s_i es el *source* y d_i es el *target* y n_i es el vector normal en el punto d_i.

En la utilización de esta última versión es necesario precomputar las normales de todos los puntos del *target* para que sea posible el cálculo del plano tangente.

4.2. Implementación del algoritmo ICP

En la Fig. 6.4 se presenta el diagrama de flujo de la localización de las nubes de puntos y la explicación de las etapas.

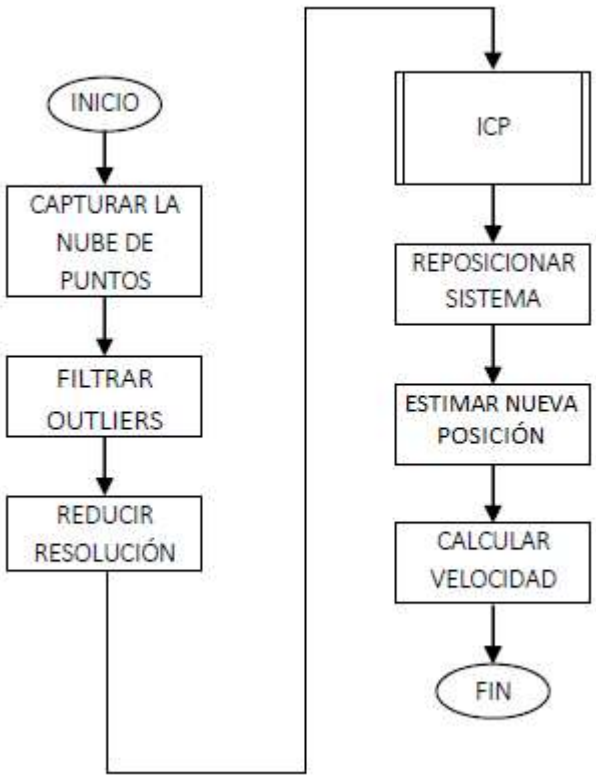


Fig. 6.4. Diagrama de flujo del proceso de localización de nube.

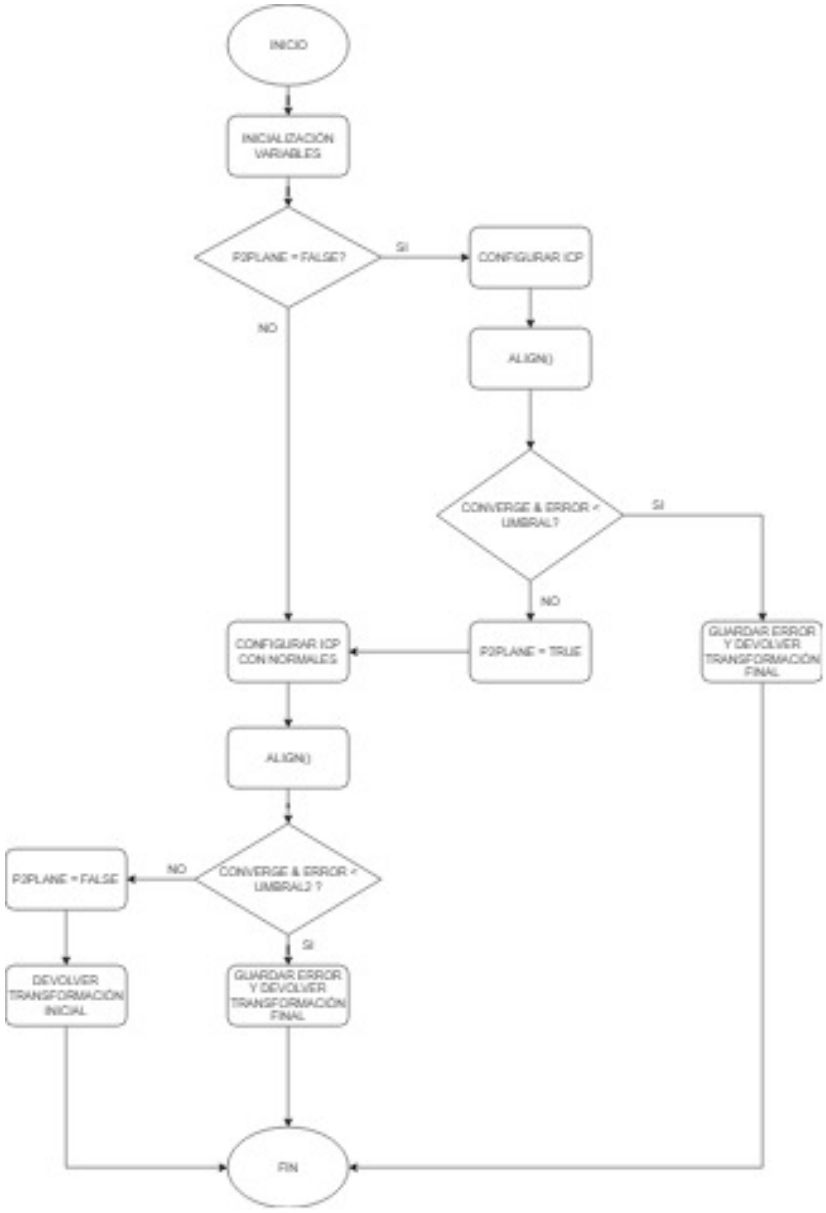


Fig. 6.5. Diagrama de flujo de la función de llamada al ICP.

En primer lugar, se carga la primera nube de puntos almacenada en la carpeta */clouds* y se almacena un puntero del tipo *pcl::PointCloud*. Destacar que el tiempo de carga de la nube no se ha tenido en cuenta a efectos de tiempos de cómputo, pero el tiempo medio de transformación de un formato a otro si, de cara al tiempo de procesamiento de cada nube.

Después se aplica un filtro que elimina los puntos NaN (Not a Number), que se producen por fallos en la obtención de datos por parte de la cámara, y también los filtros ya mencionados anteriormente, *pcl::VoxelGridFilter*, *pcl::StatisticalOutlierRemoval* y *pcl::RadiusOutlierRemoval*.

Después se procede a llamar al ICP. La función toma como parámetros de entrada la nube correspondiente al *target* y una matriz que es la estimación de la posición en la que va a estar. Inicialmente esta matriz está inicializada a la matriz origen establecida en el archivo *origin.txt*.

La función encargada de aplicar el ICP se detalla en la Fig. 6.5. Inicialmente, se establece a *false* el parámetro *p2plane*, en la creación del mapa. Después se aplica el punto a punto o punto a plano en función de este parámetro. Si el resultado converge y está dentro de un margen de error se devuelve la transformación devuelta por el ICP.

Este algoritmo se ha implementado de tal manera que el algoritmo principal de aplicación sea el punto a punto y en caso de no obtener unos resultados óptimos, habilitar la ejecución del punto a plano con unos umbrales de error un poco más amplios que en el caso anterior para favorecer la localización. En caso de que, aun así, no se obtenga un resultado aceptable, se devuelve la transformación que se había introducido como parámetro de entrada para evitar desajustes.

Dentro de la librería *pcl::IterativeClosestPoint* tenemos muchas opciones de configuración para afinar la obtención de la matriz de transformación entre la última posición y la actual.

Cuenta con tres criterios por los cuales el algoritmo terminará:

1. El número de iteraciones establecido.
2. El épsilon entre la transformación previa y la transformación estimada actual es menor que la que el usuario ha impuesto.
3. La suma de los errores cuadráticos euclídeos es más pequeña que el rango preestablecido por el usuario.

Éstos, y algunos otros que se indican a continuación, son los parámetros utilizados en este proyecto:

- setMaximumIterations : Establece el número máximo de iteraciones que queremos que compute el algoritmo (criterio 1). Esta función se hereda de la clase *pcl::Registration*.
- setTransformationEpsilon: Configura un épsilon límite entre las distancias de la transformación anterior y la actual (criterio 2). Esta función se hereda de la clase *pcl::Registration.*
- setEuclideanFitnessEpsilon: Configura un épsilon límite entre las distancias euclídeas de los puntos del *source* y del *target* (criterio 3). Esta función se hereda de la clase *pcl::Registration.*
- setInputSource: Establece el *source*.
- setInputTarget: Establece el *target*.
- setMaxCorrespondanceDistance: Fija una distancia máxima entre las correspondencias que se han establecido previamente. Esta función se hereda de la clase *pcl::Registration.*
- setTransformationEstimation: Establece el tipo de objeto que define el cálculo de las tranformaciones (En nuestro caso, o punto a punto o punto a plano). Esta función se hereda de la clase *pcl::Registration.*
- getFitnessScore: Entrega el error en las distancias euclídeas de ambas nubes de puntos. Esta función se hereda de la clase *pcl::Registration.*
- hasConverged: Devuelve si el algoritmo ha convergido o no. Esta función se hereda de la clase *pcl::Registration.*
- align: Toma como entrada dos parámetros, el *target* y una matriz, y se encarga de alinear el *target* y el *source* teniendo en cuenta toda la parametrización utilizada. Almacena en la matriz introducida la transformación final que relaciona ambas nubes, en caso de haber convergido. Esta función se hereda de la clase *pcl::Registration*.

Una vez que el ICP nos ha entregado la matriz de transformación entre la posición actual y la anterior, se reposiciona el sistema y se estima la siguiente posición.

El cálculo de la siguiente posición se realiza mediante la proyección del vector de movimiento, descrito por la posición anterior y la actual, sobre la posición actual, es decir, se está aplicando una estimación lineal de la posición.

Después se actualizaría la posición anterior y con esta información el bucle volvería a ejecutarse.

5. EJECUCIÓN DEL PROGRAMA

La ejecución del programa *tfg_processor* es configurable desde el terminal. A continuación se muestran las opciones disponibles que se muestran al mostrar el mensaje de ayuda:

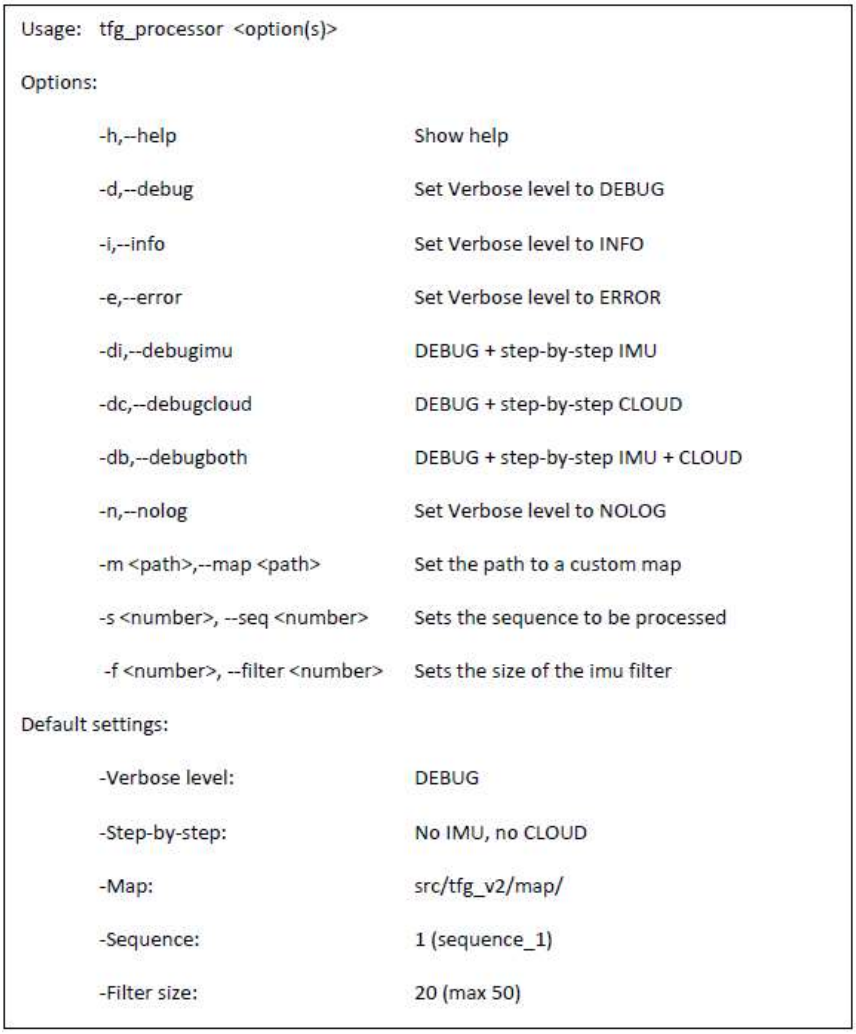


Fig. 6.6. Mensaje de ayuda de *tfg_processor*.

CAPÍTULO 7

Pruebas de funcionamiento

En este capítulo se muestran los resultados obtenidos tras ejecutar el programa *tfg_processor* sobre las secuencias transformadas con *tfg_recorder*. Se presentan un total de 3 secuencias en las que se ha buscado probar la eficacia del algoritmo ICP en distintos entornos y con distinta parametrización.

Destacar que el conjunto de todas las pruebas realizadas en este capítulo se han realizado sobre un portátil de la marca *Dell*, modelo Latitude E6440, con 3.8Gb de memoria RAM, un procesador Intel® Core™ i5-4200M a 2.5GHz con 4 núcleos.

1. SECUENCIA 1

En esta secuencia se trabajará sobre un entorno que es el interior de una vivienda, concretamente una habitación de la misma, en la que hay diferentes elementos del mobiliario, como pueden ser estanterías, mesas, ordenadores y monitores. Con esta primera secuencia lo que se busca es ver si el algoritmo responde bien a desplazamientos lineales cuando el nivel de detalle de la escena es alto. Supone un buen punto de partida para intentar establecer una parametrización base del algoritmo.

El recorrido trazado con la cámara ha sido un movimiento lineal de aproximadamente un metro de distancia. El mapa de la escena se puede observar a continuación:

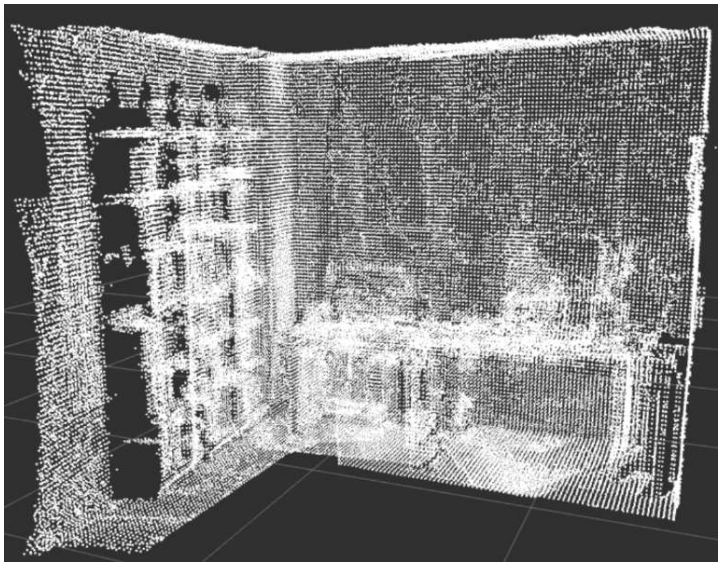


Fig. 7.1. Mapa de la secuencia 1.

1.1. Análisis secuencia 1

El resultado obtenido tras realizar el procesamiento completo de la secuencia sobre el mapa es el siguiente:

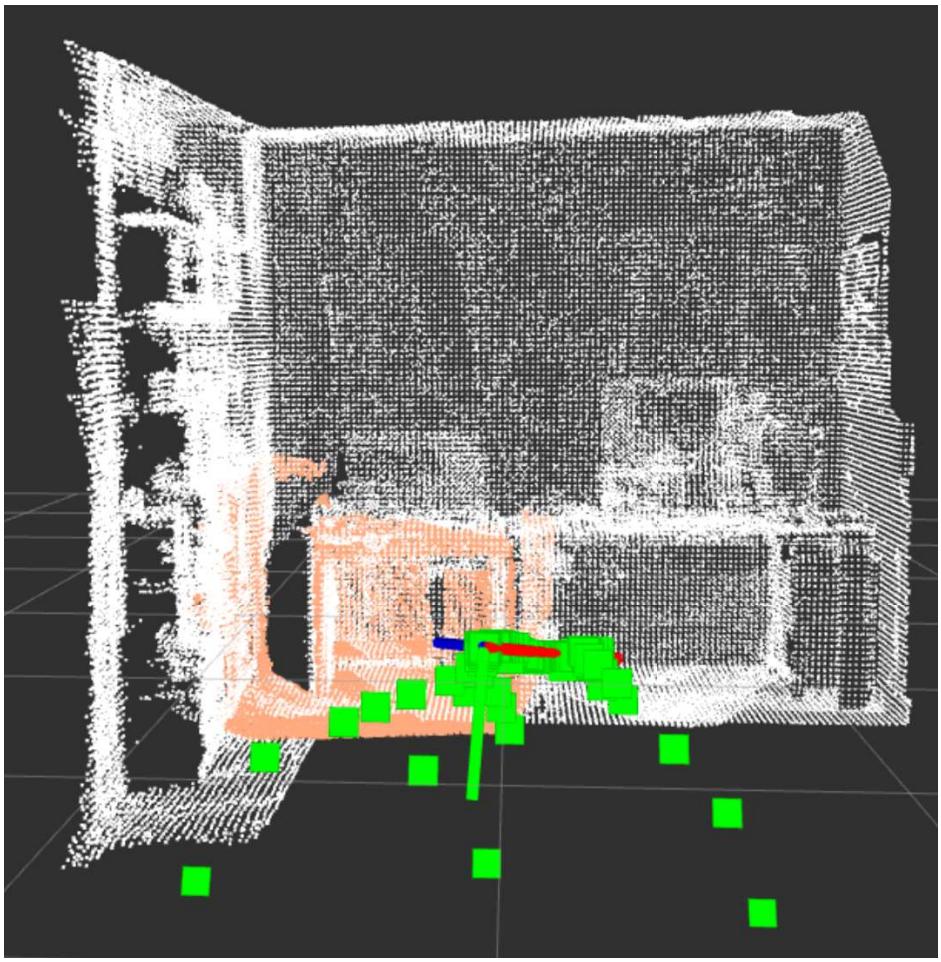
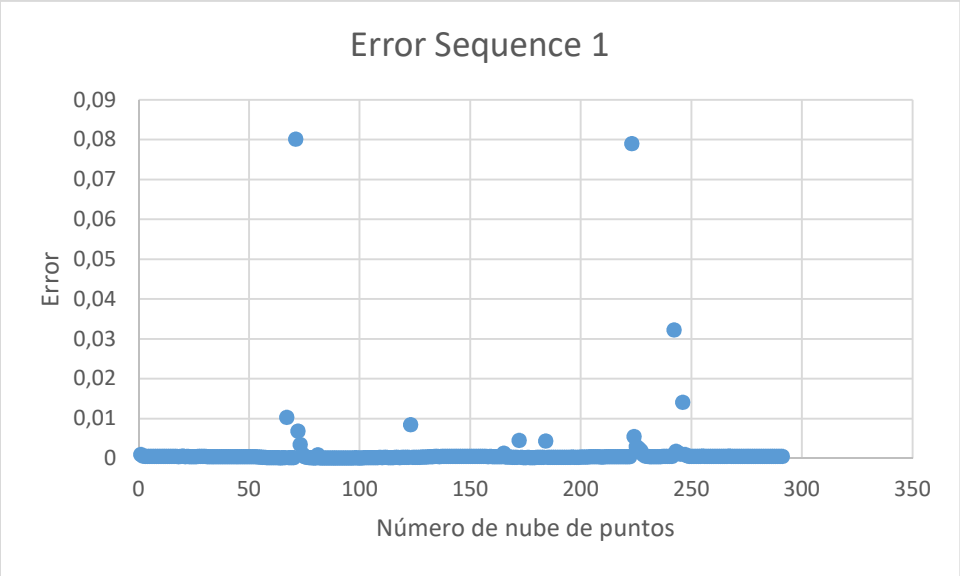


Fig. 7.2. Procesamiento completo de la secuencia 1.

Sobre el mapa podemos observar la última nube de puntos procesada en un color anaranjado, al igual que la última referencia de ésta. Los cuadrados verdes representan cada una de las posiciones que ha encontrado el ICP para cada nube. Como podemos ver, existen algunos puntos que se encuentran alejados del grueso de muestras.

Estos puntos se corresponden con algunas nubes de puntos que el algoritmo no ha logrado localizar correctamente, pero como se puede observar, ha corregido la posición en varias ocasiones logrando posicionar de nuevo el localización correcta.

Se ha realizado un gráfico en el que se muestra el error en las distancias euclídeas correspondiente a cada nube:



Gráfica de error de cada nube de puntos.

En total había 291 nubes de puntos correspondientes a la secuencia 1. Entre 10 y 12 de ellas tienen un error que se aleja de la media. Esto supone entre un 3.5% y un 4.2% de tasa de error, es decir, nos estamos moviendo en márgenes aceptables de error.

Lo que se ha observado en esta primera secuencia es que la variante de punto a punto del ICP ha convergido en todos los casos, y ha localizado bastante bien las nubes, lo que ha hecho innecesario que entre la segunda variante que se había implementado, la de punto a plano.

1.2. Análisis de tiempos

En este apartado se van a comentar los resultados obtenidos en la secuencia 1 tanto de los tiempos de captura como de procesamiento de la cámara.

Para esta primera secuencia se ha obtenido un período en la captura de datos de 47 ms, esto equivale a 21 Hz. Como ya se mencionó, estos datos son virtuales, ya que la captura de datos se ha llevado a cabo a través de una grabación reproducida más lentamente. Aun así, ha permitido la captura de todos los mensajes, que era lo que se buscaba.

Los tiempos de procesamiento se recogen en la siguiente tabla:

	Time (s)
Average Cloud processing time	**0.570603**
MAX Cloud processing time	**0.710902**
MIN Cloud processing time	**0.505566**

Como se observa, se tiene medio segundo de procesamiento por cada nube, lo que es un tanto elevado.

Si calculamos el ratio de ocupación, y tenemos en cuenta que la cámara está configurada para trabajar a unos 30Hz, obtenemos lo siguiente:

$$ratio_{cam} = \frac{tiempo_{cómputo}}{período} = \frac{0.570603s}{33ms}\left(\frac{1000ms}{1s}\right) = 17,1180 \rightarrow 1711,80\% \qquad (7.1.1)$$

Como se puede ver, son ratios que no propician la implementación como aplicación de tiempo real.

2. SECUENCIA 2

En esta secuencia se trabaja sobre el mismo entorno anterior pero buscando probar cómo responde la estimación lineal que se ha implementado a la hora de proporcionar una transformación previa al ICP. Para ello se ha trazado una trayectoria de subida, y en un momento determinado se ha cambiado repentinamente a una trayectoria de bajada.

2.1. Análisis secuencia 2

En la Fig. 7.3 podemos observar el entorno con puntos blancos, que es el mismo que el anterior, la última captura procesada en color anaranjado y en verde el recorrido que ha seguido la cámara. El color verde en la trayectoria indica que sólo se ha ejecutado el algoritmo de punto a punto.

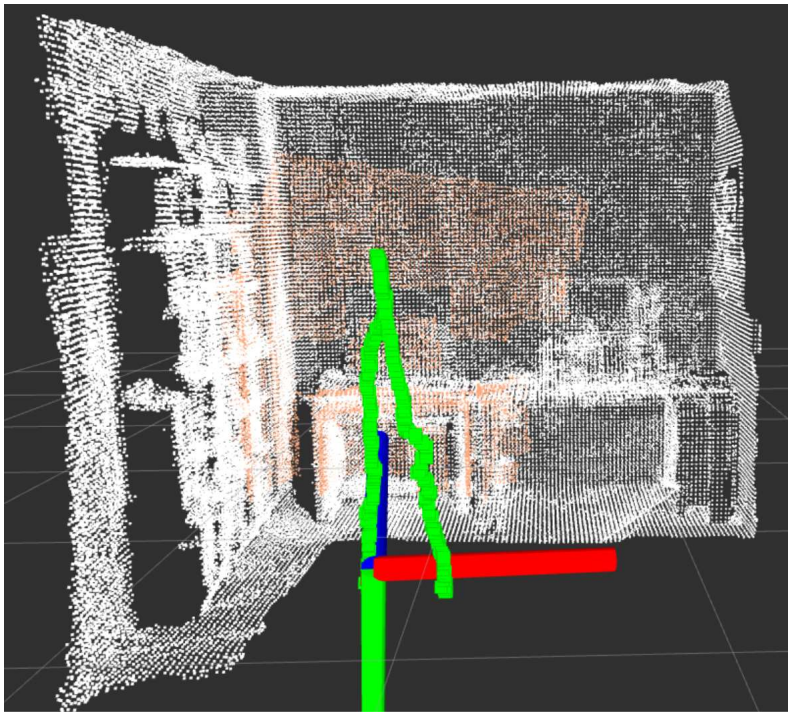


Fig. 7.3. Procesamiento completo de la secuencia 2.

En este procesamiento se puede observar una trayectoria mucho más limpia que en el anterior.

En los puntos más altos de la trayectoria se encuentran zonas planas con pocos detalles y aun así el punto a punto ha logrado alinear correctamente las nubes.

La parametrización ha sido parecida a la secuencia 1, se ha aplicado una reducción en la resolución utilizando el mencionado *VoxelGrid Filter* creando cubos de 2.5 cm de lado. No se ha detectado la necesidad de aplicar el filtro de *outliers*.

El número de iteraciones máximas se ha tenido que aumentar a 50 para favorecer la localización y que converja el algoritmo.

El error en las distancias euclídeas se muestra a continuación:

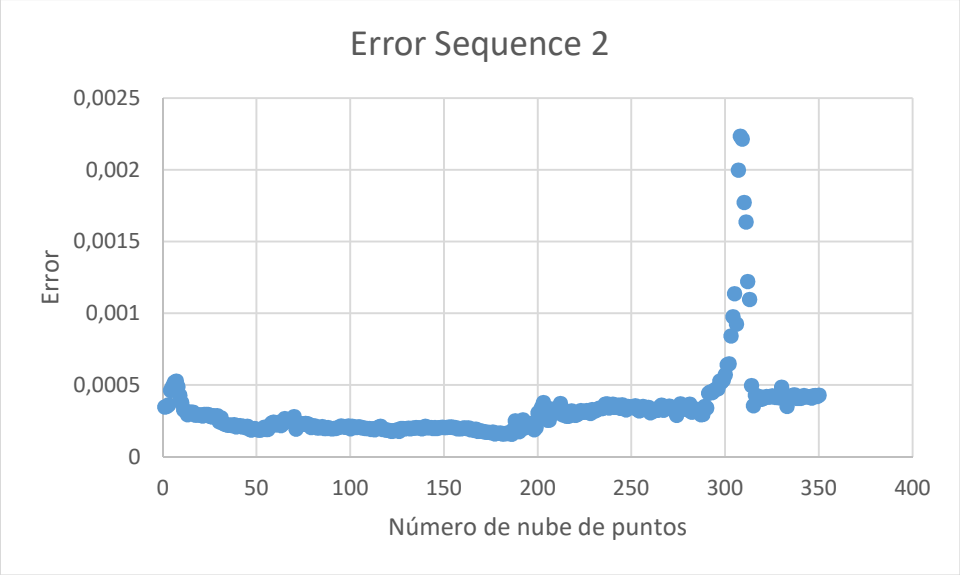


Gráfico de error de cada nube de puntos.

Se puede observar que el error se mantiene constante, exceptuando una zona al final de la secuencia. A pesar de este pequeño repunte, el error no excede el umbral establecido para el cambio de algoritmo, que era de 0,003 m. En general se han obtenido unos resultados más aceptables que en la secuencia anterior.

2.2. Análisis de tiempos

Los tiempos en la captura de la secuencia con el programa *tfg_recorder* son de 44 ms de período, es decir, 22Hz. Frecuencia adecuada para permitir la captura de todos los mensajes enviados por la cámara.

En los tiempos de procesamiento podemos esperar un incremento notable debido a que se han aumentado las iteraciones máximas que puede realizar el ICP.

En la siguiente tabla se recogen los tiempos de procesamiento obtenidos para esta segunda secuencia:

	Time (s)
Average Cloud processing time	**3.160327**
MAX Cloud processing time	**4.277898**
MIN Cloud processing time	**1.751699**

Como se puede observar los tiempos de procesamiento se han elevado demasiado, pero, en cambio, se ha obtenido una localización más precisa. Este va a ser uno de los problemas más importantes a la hora de ajustar los parámetros del ICP. Es necesario llevar a un equilibrio entre conseguir una localización aceptable y unos tiempos de cómputo bajos.

Calculando el ratio de ocupación de la CPU debido a las nubes de puntos, si sólo tuviera un núcleo:

$$ratio_{cam} = \frac{tiempo_{cómputo}}{período} = \frac{3{,}160327s}{33ms}\left(\frac{1000ms}{1s}\right) = 95{,}7675 \rightarrow 9576{,}75\% \qquad (7.2.1)$$

A nivel de tiempos de procesamiento, es un resultado nada aceptable. En la siguiente secuencia se va tratar de equilibrar esta situación.

3. SECUENCIA 3

En esta secuencia tratamos otro entorno. Se ha buscado una zona con tramos planos, y algún detalle mínimo para probar así el punto a plano. En la Fig. 7.4 se puede observar el mapa de este entorno.

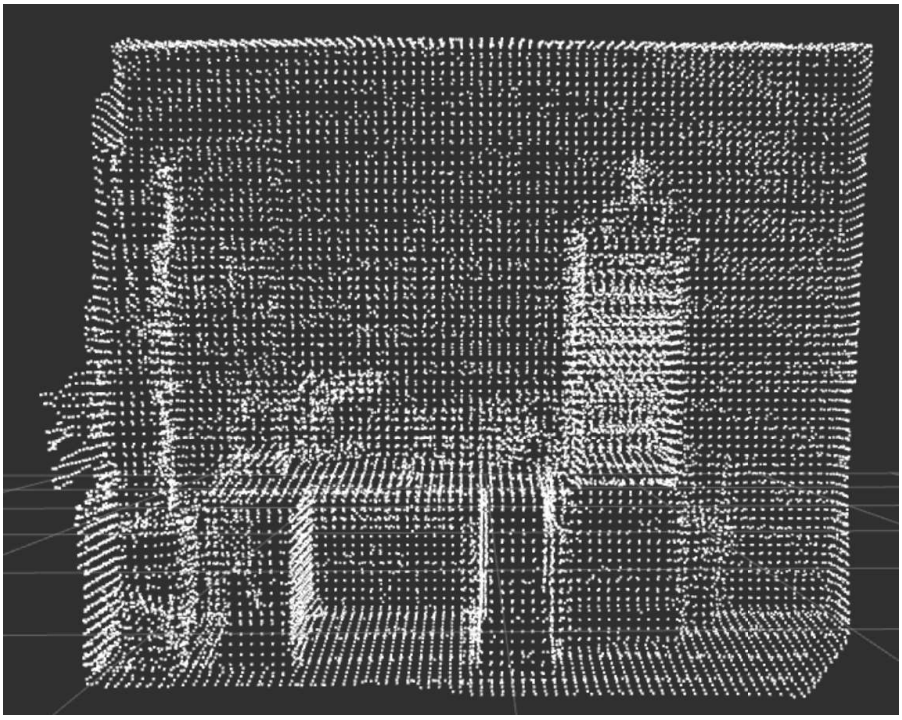


Fig. 7.4. Mapa utilizado en la secuencia 3.

3.1. Análisis Secuencia 3

En esta secuencia se han cambiado bastantes cosas respecto a las anteriores. El filtro que reduce la resolución se ha aumentado considerablemente, hasta un tamaño de cubo de 4 cm por lado. Sí se han aplicado los filtros de eliminación de *outliers*. El umbral del punto a punto se ha establecido en 0,001 y el del punto a plano en 0,003.

El número de iteraciones máximas se ha mantenido en 50. En la Fig. 7.5 se muestra el procesamiento completo de esta secuencia.

A primera vista se puede observar que en este caso sí que ha habido alternancia entre los métodos de punto a punto y punto a plano, ya que existen puntos verdes y puntos rojos en la trayectoria obtenida.

Los puntos verdes corresponden con las localizaciones obtenidas con el punto a punto y las rojas con el punto a plano.

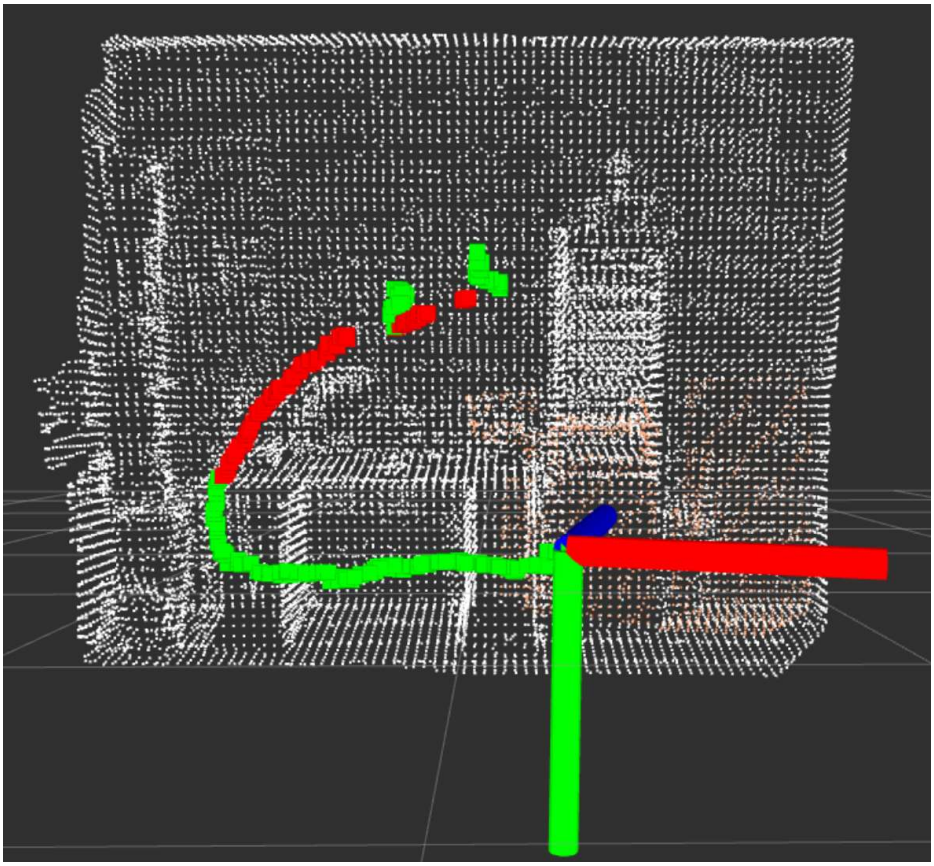


Fig. 7.5. Procesamiento completo de la secuencia 3.

La trayectoria obtenida en la Fig. 7.5 se corresponde con mucha precisión a la descrita con el movimiento real de la cámara. Pero existen tramos en los que se observan pequeños saltos.

Estos saltos son debidos a que se estaba llegando a zonas planas y todavía estaba en ejecución el punto a punto, y se ha observado que en este entorno no ha respondido muy bien.

Cuando el error ha crecido lo suficiente y ha dado paso al punto a plano sí que se ha visto un pequeño salto que ha posicionado la nube casi perfectamente sobre su correspondiente zona en el mapa. En este aspecto, el punto a plano logra orientar las nubes correspondientes a zonas planas de manera excepcional.

Ha sido gracias al punto a plano por lo que se ha obtenido una trazado final muy aceptable.

Se puede decir también que en el caso de tener zonas planas con poco detalle el punto a punto no ofrece resultados del todo buenos. En comparación con la secuencia anterior, en este caso tenemos planos perfectos, sin ningún tipo de esquinas o detalles importantes que ofrezcan cambios bruscos en el relieve, por esta razón el punto a punto no ha ofrecido los mismos resultados que anteriormente.

3.2. Análisis de tiempos

En este caso se han mantenido las iteraciones máximas del ICP pero se ha reducido considerablemente la resolución del mapa y de las capturas individuales provenientes de la cámara. Esto supone un factor muy importante ya que contra menos puntos existen menos tardará en recorrerlos.

El mapa grabado inicialmente contaba con 4.047.973 puntos. Al aplicar el filtro con cubos de 4 cm de lado se ha reducido a 20.526 puntos. Esto se espera que tenga un impacto positivo en la reducción de tiempos.

El tiempo de captura con el programa *tfg_recorder* ha sido de 40 ms de período, 25Hz de frecuencia.

Siguen siendo tiempos que permiten la captura de todos los mensajes.

Los tiempos de procesamiento se recogen en la siguiente tabla:

	Time (s)
Average Cloud processing time	**0.523270**
MAX Cloud processing time	**1.456747**
MIN Cloud processing time	**0.108801**

Se observa un cambio positivo. Los tiempos de procesamiento han bajado, pero aún sigue suponiendo un problema ya que son altos.

Calculando el ratio de ocupación como en los casos anteriores podemos obtener:

$$ratio_{cam} = \frac{tiempo_{cómputo}}{período} = \frac{0,523270s}{33ms}\left(\frac{1000ms}{1s}\right) = 15,8567 \rightarrow 1585,67\% \qquad (7.3.1)$$

Se sigue obteniendo un ratio de ocupación muy elevado.

Se necesitaría encontrar una manera de realizar el procesamiento con varios procesos trabajando en paralelo y seguir aplicando otra parametrización que favorezca la reducción de tiempos de procesamiento.

CAPÍTULO 8

Conclusiones

A pesar de haber logrado localizar cada una de las secuencias con una buena precisión, existen ciertos aspectos que son necesarios mejorar de cara a futuros proyectos.

En el programa *tfg_recorder* se ha tenido que hacer uso de la aplicación *rosbag* para el grabado de las secuencias debido a que los tiempos de guardado de las nubes de puntos son demasiado elevados y no permitían la captura de todos los mensajes.

Se podría estudiar aplicar el filtrado antes de guardar las nubes de puntos, pero el tiempo de filtrado también habría que añadirlo y ver si supone una mejora respecto de la situación de no filtrar.

Un aspecto positivo que se ha observado en las pruebas realizadas es que la conmutación entre el punto a punto y el punto a plano ha proporcionado una buena versatilidad al programa. El uso de diferentes tipos de técnicas se verá necesario conforme los entornos se vayan haciendo más complejos y con menos detalles, por lo que seguir investigando en las opciones que implementa de manera continua la librería PCL es una vía de solucionar estos problemas emergentes.

El otro gran problema que han sacado a la luz las pruebas de funcionamiento son los tiempos de cómputo de las nubes. Se puede aumentar el filtro aplicado a las mismas, pero siempre llegará un punto en el que ya no se podrá filtrar más, haciendo que esta tarea dependa totalmente del hardware sobre el que se ejecuta.

Actualmente existen equipos investigando en el registro en paralelo de nubes de puntos [16]. También se están desarrollando técnicas de procesamiento en paralelo con matrices de procesadores. Pero estas tecnologías están de momento en desarrollo.

Otra manera de redecir el ratio de utilización de la CPU sería integrar un IMU en el proceso de localización para poder tomar menos muestras con la cámara. Pero habría que tener en cuenta que el IMU entrega aceleraciones angulares, y eso sería necesario convertirlo a posición, por lo que una doble integración puede resultar fatal para el crecimiento de errores.

Por ello, un IMU podría ayudar a aliviar carga pero sería necesario investigar muy bien su integración.

Bibliografía

[1] G. Elbaz, T. Avraham y A. Fischer, «3D Point Cloud Registration for Localization using a Deep Neural Network Auto-Encoder.,» *Technion - Israel Institute of Technology,* 2017.

[2] B. Bellekens, V. Spruyt, R. Berkvens, R. Penne y M. Weyn, «A Benchmark Survey of Rigid 3D Point Cloud Registration Algorithms,» *International Journal on Advances in Intelligent Systems,* vol. VIII, nº 1 & 2, 2015.

[3] D. Turbica Mamblona, «Sistema de localización con cámara RGBD basado en ICP sobre un entorno conocido,» Zaragoza, 2017.

[4] S. Mattoccia y M. Poggi, «Department of Computer Science and Engineering (DISI),» 2015. [En línea]. Available: https://vision.disi.unibo.it/~mpoggi/papers/icdsc2015.pdf. [Último acceso: 23 Julio 2019].

[5] «Portal Online de ASUS,» [En línea]. Available: https://www.asus.com/es/3D-Sensor/Xtion_PRO_LIVE/overview/. [Último acceso: 23 Julio 2019].

[6] «Robot Operating System. Página web oficial.,» Open Source Robotics Foundation, [En línea]. Available: http://www.ros.org. [Último acceso: 4 Agosto 2019].

[7] «PCL Official Site,» [En línea]. Available: http://pointclouds.org/. [Último acceso: 20 Agosto 2019].

[8] «ROS Documentation,» [En línea]. Available: https://wiki.ros.org/rgbdslam. [Último acceso: 20 Agosto 2019].

[9] F. Endres, J. Hess, J. Sturm, D. Cremers y W. Burgard, «3D Mapping with an RGB-D Camera,» *IEEE TRANSACTIONS ON ROBOTICS,* vol. 30, nº 1, p. 11, 2014.

[10] «RGBDSlam V2 Github page,» [En línea]. Available: https://felixendres.github.io/rgbdslam_v2/. [Último acceso: 20 Agosto 2019].

[11] M. Labbé y F. Michaud, «Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation,» p. 12, 2013.

[12] R. Kuemmerle, G. Grisetti, H. Strasdat, K. Konolige y W. Burgard, «Official Github page of g2o - General Graph Optimization,» [En línea]. Available: https://github.com/RainerKuemmerle/g2o. [Último acceso: 31 Agosto 2019].

[13] «Official ROS Wiki,» [En línea]. Available: http://wiki.ros.org/rosbag/Commandline. [Último acceso: 1 Septiembre 2019].

[14] W. Burgard, C. Stachniss, M. Bennewitz y K. Arras, «Department of Autonomous Intelligent Systems. University of Freiburg.,» [En línea]. Available: http://ais.informatik.uni-

freiburg.de/teaching/ss12/robotics/slides/17-icp.pdf. [Último acceso: 6 Septiembre 2019].

[15] K.-L. Low, «Linear Least-Squares Optimization for Point-to-Plane ICP Surface Registration,» Department of Computer Science, University of North Carolina, Chapel Hill, 2004.

[16] H. Lu y Y. Wei , «Parallel Point Cloud Registration,» 2016. [En línea]. Available: https://github.com/HanzhouLu/Parallel-Point-Cloud-Registration/blob/master/Final%20Report.pdf. [Último acceso: 16 Septiembre 2019].

[17] D. Tedaldi, «IMU calibration without mechanical equipment,» *Master's Thesis. Dipartimento di Ingegneria dell'Informazion. Universitá degli Studi di Padova,* 23 Septiembre 23 de septiembre de 2013.

[18] «PHIDGETS INC,» 2016. [En línea]. Available: https://www.phidgets.com/?tier=3&catid=10&pcid=8&prodid=1158. [Último acceso: 30 Julio 2019].

ANEXOS

ANEXO 1. SENSOR_MSGS/POINTCLOUD2 63
ANEXO 2. TFG_RECORDER.H 64
ANEXO 3. TFG_LOGGER.H 67
ANEXO 4. TFG_RECORDER.CPP 69
ANEXO 5. TFG_SET_ORIGIN.CPP 73
ANEXO 6. TFG_CLOUD.H 77
ANEXO 7. TFG_CLOUD.CPP 79
ANEXO 8. TFG_CLOUDMSG.H 83
ANEXO 9. TFG_CLOUDMSG.CPP 85
ANEXO 10. TFG_DRONE.H 87
ANEXO 11. TFG_DRONE.CPP 91
ANEXO 12. TFG_FILTER.H 93
ANEXO 13. TFG_FILTER.CPP 95
ANEXO 14. TFG_FILTERVECTOR3.H 97
ANEXO 15. TFG_FILTERVECTOR3.CPP 99
ANEXO 16. TFG_LOGGER.H 101
ANEXO 17. TFG_MAP.H 103
ANEXO 18. TFG_MAP.CPP 105
ANEXO 19. TFG_PROCESS.H 109
ANEXO 20. TFG_PROCESS.CPP 111
ANEXO 21. TFG_PROCESSOR.H 113
ANEXO 22. TFG_PROCESSOR.CPP 115
ANEXO 23. TFG_TIMER.H 129
ANEXO 24. TFG_TIMER.CPP 131

ANEXO 25. TFG_UTILITIES.H 133

ANEXO 26. TFG_UTILITIES.CPP 135

Anexo 1. sensor_msgs/PointCloud2

sensor_msgs/PointCloud2	
std_msgs/Header	header
uint32	height
uint32	width
sensor_msgs/PointField[]	fields
bool	is_bigendian
uint32	point_step
uint32	row_step
uint8[]	data
bool	is_dense

std_msgs/Header	
uint32	seq
time	stamp
string	frame_id

sensor_msgs/PointField		
uint8	INT8	=1
uint8	UINT8	=2
uint8	INT16	=3
uint8	UINT16	=4
uint8	INT32	=5
uint8	UINT32	=6
uint8	FLOAT32	=7
uint8	FLOAT64	=8
string	name	
uint32	offset	
uint8	datatype	

Anexo 2. tfg_recorder.h

```
//=============================================================================
//
// File: tfg_recorder.h
//  Author: David Turbica, Ramiro Costa
//
//=============================================================================

#ifndef __TFG_RECORDER_H_INCLUDED__
#define __TFG_RECORDER_H_INCLUDED__

#include <ros/ros.h>
#include <signal.h>
#include <ctime>


//C++ includes

#include <string>
#include <boost/thread/mutex.hpp>
#include <boost/foreach.hpp>
#include <rosbag/view.h>

//ROS includes
#include <sensor_msgs/PointCloud2.h>
#include <rosbag/bag.h>

//PCL includes
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>

#include "tfg_logger.h"

//TFG includes

//Global Variables
int instant; //in miliseconds
long initial_instant; //in miliseconds
bool initial_instant_set; //in miliseconds
const std::string path_clds = "src/tfg/tfg_recorder/sequence/clouds/";
const std::string path_clds_data
= "src/tfg/tfg_recorder/sequence/clouds_data.txt";

const std::string path_parameter
= "src/tfg/tfg_recorder/sequence/parameters.txt";


//Variables CAM
long cld_t;
int cld_ratio;
```

```
int cld_counter;
std::ofstream cld_file;

//Functions
int main (int argc, char** argv);
void processCam (const sensor_msgs::PointCloud2::ConstPtr &input);
void customSigintHandler(int sig);


#endif //__TFG_RECORDER_H_INCLUDED
```

Anexo 3. tfg_logger.h

```
//==================================================================
//
// File: tfg_logger.h
//  Author: David Turbica, Ramiro Costa
//
//==================================================================

#ifndef __TFG_LOGGER_H_INCLUDED__
#define __TFG_LOGGER_H_INCLUDED__

#include <iostream>

//Logging levels:
// - DEBUG 0 (GREEN) [32]
//  - INFO 1 (BLUE) [34]
//  - ERROR 2 (RED) [31]
//  - NOLOG     3

static int LOG_LEVEL = 0;

#define PRINT_ENTER(lines) for(int i=0;i<lines;i++) std::cout <<
std::endl;

#define PRINT_DEBUG(line) if(LOG_LEVEL<=0)
std::cout<< "\033[32m" <<line<< "\033[0m" <<std::endl;

#define PRINT_INFO(line)  if(LOG_LEVEL<=1)
std::cout<< "\033[34m" <<line<< "\033[0m" <<std::endl;

#define PRINT_ERROR(line) if(LOG_LEVEL<=2)
std::cout<< "\033[31m" <<line<< "\033[0m" <<std::endl;

#endif //__TFG_LOGGER_H_INCLUDED
```

Anexo 4. tfg_recorder.cpp

```
//=================================================================================
//
// File: tfg_recorder.cpp
//  Author: David Turbica, Ramiro Costa
//
//=================================================================================

#include "tfg_recorder/tfg_recorder.h"

double t_cld = 0.0;


int main (int argc, char** argv) {
//---------------------------------------------------------------------------
// Main function. It initializes the whole program and sets the timers and
//  and topics' subscribers and publishers.
//---------------------------------------------------------------------------

PRINT_INFO
("=================================================================================");
PRINT_INFO ("|                                                                               |");
PRINT_INFO ("|                        TFG_RECORDER                                           |");
PRINT_INFO ("|                                                                               |");
PRINT_INFO
("|===============================================================================|");
PRINT_INFO ("|                                                                               |");
PRINT_INFO ("| Author:          David Turbica, Ramiro Costa                                  |");
PRINT_INFO ("|                                                                               |");
PRINT_INFO ("| Description:      It records a sequence of PointCloud2                        |");
PRINT_INFO ("|                  ROS-messages to a file so it can be used as                  |");
PRINT_INFO ("|                  input data later on.                                         |");
PRINT_INFO ("|                                                                               |");
PRINT_INFO
("=================================================================================");

//Initialize ROS
ros::MultiThreadedSpinner spinner(2);
ros::init (argc, argv, "tfg_recorder");
ros::NodeHandle nh;
ros::NodeHandle nh_imu;
signal(SIGINT, customSigintHandler);
LOG_LEVEL = 0;

//Initialize global variables
instant = 0;
initial_instant = 0;
initial_instant_set = false;

//Initialize CAM variables
cld_ratio = 0;
cld_counter = 1;
cld_file.open (path_clds_data.c_str());

//Subscribers
```

```
ros::Subscriber sub_cam = nh.subscribe ("/camera/depth/points", 1, processCam);

PRINT_INFO ("Node tfg_core is set up and running");

//Spin
spinner.spin();
}

void processCam (const sensor_msgs::PointCloud2::ConstPtr &input) {
//---------------------------------------------------------------------------
// Function called when a new message from '/camera/depth/points' topic is
//  received.
//---------------------------------------------------------------------------
PRINT_DEBUG ("Processing cloud " << cld_counter);
//Check for the initial time

if (!initial_instant_set){
long t = (long)input->header.stamp.sec * 1000 + (input->header.stamp.nsec / 1000000) -
initial_instant;
initial_instant = t;
initial_instant_set = true;
}

//Save the cloud
pcl::PointCloud<pcl::PointXYZ> cloud;
std::clock_t start = std::clock();
pcl::fromROSMsg (*input, cloud);
std::clock_t end = std::clock();
double elapsed_secs = double(end - start) / CLOCKS_PER_SEC;
printf("Time: %f\n", elapsed_secs);
pcl::io::savePCDFile (path_clds + "cloud_" + boost::to_string(cld_counter) + ".pcd", cloud);
//Save the time stamp
long t = (long)input->header.stamp.sec * 1000 + (input->header.stamp.nsec / 1000000) -
initial_instant;
cld_file << "cloud_" + boost::to_string(cld_counter) + ".pcd " + boost::to_string(t) + "\n";
//Calc the average rate
cld_ratio = cld_ratio + (((int)(t - cld_t) - cld_ratio) / cld_counter);
cld_t = t;
cld_counter ++;
}

void customSigintHandler(int sig){
//---------------------------------------------------------------------------
// Function that overrides default ROS sigint handler. It allows to print
//  some information when the node has been shuted down and end child
//  processes if needed.
//---------------------------------------------------------------------------
//Closing files oppened
if (cld_file.is_open()) cld_file.close();
//Writing parameters file
std::ofstream param_file;
param_file.open (path_parameter.c_str());
if (param_file.is_open()){
// We subtract 2 in order to prevent the latest data from not being written entirely to file.
param_file << "Cld_instances: " << cld_counter - 2 << "\n";
//Closing the file
param_file.close();
```

```
}
PRINT_INFO ("");
PRINT_INFO ("Cam   period   " << cld_ratio << " ms");
PRINT_INFO ("      ratio    " << 1000/cld_ratio << " Hz");
PRINT_INFO ("Imu   period   " << imu_ratio << " ms");
PRINT_INFO ("      ratio    " << 1000/imu_ratio << " Hz");
PRINT_INFO ("");
PRINT_INFO
("=============================================================================");
PRINT_INFO ("|                                                                           |");
PRINT_INFO ("|                        TFG_RECORDER                                       |");
PRINT_INFO ("|                                                                           |");
PRINT_INFO
("|===========================================================================|");
PRINT_INFO ("|                                                                           |");
PRINT_INFO ("| Author:           David Turbica, Ramiro Costa                             |");
PRINT_INFO ("|                                                                           |");
PRINT_INFO ("| Description:      It records a sequence of PointCloud2 and IMU            |");
PRINT_INFO ("|                   ROS-messages to a file so it can be used as             |");
PRINT_INFO ("|                   input data later on.                                    |");
PRINT_INFO ("|                                                                           |");
PRINT_INFO
("|===========================================================================|");
PRINT_INFO ("|                                                                           |");
PRINT_INFO ("| Node terminated:  No reason given (Ctrl+c)                                |");
PRINT_INFO ("|                                                                           |");
PRINT_INFO
("=============================================================================");
//Overrides ROS default Sigint Handler
ros::shutdown();
}
```

Anexo 5. tfg_set_origin.cpp

```
//===================================================================
//
// File: tfg_set_origin.cpp
//  Author: Ramiro Costa
//
//===================================================================

#include "iostream"
#include "ros/ros.h"
#include "sensor_msgs/PointCloud2.h"
#include "pcl_conversions/pcl_conversions.h"
#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/point_cloud.h>
#include <pcl/console/parse.h>
#include <pcl/common/transforms.h>
#include <pcl/registration/icp.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/point_types.h>
#include <tfg_set_origin/tfg_utilities.h>
#include <geometry_msgs/Transform.h>
#include <geometry_msgs/PoseStamped.h>

int
main (int argc, char** argv)
{
ros::init(argc, argv, "tfg_set_origin");
ros:: NodeHandle n;

//pcl::PCLPointCloud<PointXYZ>
pcl::PointCloud<pcl::PointXYZ>::Ptr source_cloud (new pcl::PointCloud<pcl::PointXYZ> ());
pcl::PointCloud<pcl::PointXYZ>::Ptr map_cloud (new pcl::PointCloud<pcl::PointXYZ> ());


//output.header.stamp = ros::Time::now();

ros::Rate loop_rate(10);

if (pcl::io::loadPCDFile<pcl::PointXYZ>("src/tfg/tfg_set_origin/src/cloud_1.pcd", *source_cloud)
== -1) //* load the file
{
PCL_ERROR ("Couldn't read file cloud_1.pcd \n");
return (-1);
}
if (pcl::io::loadPCDFile<pcl::PointXYZ>("src/tfg/tfg_set_origin/src/mapa_inverso_cuarto.pcd", *m
ap_cloud) == -1) //* load the file
{
PCL_ERROR ("Couldn't read file cloud_1.pcd \n");
return (-1);
}
pcl::PointCloud<pcl::PointXYZ>::Ptr map_filtered (new pcl::PointCloud<pcl::PointXYZ> ());
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl::PointXYZ> ());
pcl::VoxelGrid<pcl::PointXYZ> filter;
filter.setInputCloud (map_cloud);
```

```
filter.setLeafSize (0.04, 0.04, 0.04);
filter.filter (*map_filtered);
pcl::VoxelGrid<pcl::PointXYZ> filter2;
filter2.setInputCloud (source_cloud);
filter2.setLeafSize(0.04, 0.04, 0.04);
filter2.filter ( *cloud_filtered );
std::vector<int> indices;
std::vector<int> indices2;
pcl::removeNaNFromPointCloud(*cloud_filtered, *cloud_filtered, indices);
pcl::removeNaNFromPointCloud(*map_filtered, *map_filtered, indices2);/**/

float theta = M_PI/2;
float theta2 = M_PI/18;

pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud (new
pcl::PointCloud<pcl::PointXYZ> ());
pcl::PointCloud<pcl::PointXYZ> aligned_cloud;

Eigen::Affine3f transform_2 = Eigen::Affine3f::Identity();
/*//transform_2.rotate (Eigen::AngleAxisf (-theta, Eigen::Vector3f::UnitY()));
//transform_2.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitZ()));
transform_2.rotate (Eigen::AngleAxisf (2*theta, Eigen::Vector3f::UnitX()));
transform_2.rotate (Eigen::AngleAxisf (theta+theta/6, Eigen::Vector3f::UnitY()));
transform_2.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitZ()));

//transform_2.rotate (Eigen::AngleAxisf (-2*theta2, Eigen::Vector3f::UnitX()));
//transform_2.rotate (Eigen::AngleAxisf (-theta2, Eigen::Vector3f::UnitZ()));
//transform_2.rotate (Eigen::AngleAxisf (theta2, Eigen::Vector3f::UnitZ()));
//transform_2.rotate (Eigen::AngleAxisf (-theta2/2, Eigen::Vector3f::UnitX()));


transform_2.translation() << 0.0, -1.2, 0.8;
std::cout << transform_2.matrix() << std::endl;
pcl::transformPointCloud (*cloud_filtered, *transformed_cloud, transform_2);
std::cout <<  "transformations done" << std::endl;
int counter=0;


pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
icp.setInputSource (transformed_cloud);
std::cout <<  "input done" << std::endl;
icp.setInputTarget (map_filtered);
std::cout <<  "map done" << std::endl;
//icp.setTransformationEpsilon (1e-8);
//icp.setEuclideanFitnessEpsilon (1);
icp.align (aligned_cloud);
std::cout <<  "aligning done" << std::endl;
Eigen::Matrix4f transformation;
std::cout <<  "final transformation done" << std::endl;

if(icp.hasConverged ()){


  std::cout <<  "TRANSFORMATIONS DONE!" << std::endl;
  transformation = icp.getFinalTransformation ();
  std::cout << transformation.matrix() << "    " << icp.getFinalTransformation() << std::endl;
} else{
```

```
PCL_ERROR ("\nICP has not converged.\n");

}
*/

ros::Publisher publisher_cloud =
n.advertise<sensor_msgs::PointCloud2>("output_cloud", 1000);
ros::Publisher publisher_map = n.advertise<sensor_msgs::PointCloud2>("output_map", 1000);
sensor_msgs::PointCloud2 output_cloud;
sensor_msgs::PointCloud2 output_map;

//output.header.stamp = ros::Time::now();

pcl::toROSMsg(*transformed_cloud,output_cloud);
pcl::toROSMsg(*map_filtered,output_map);
std::cout << map_cloud->points.size() << std::endl;
std::cout << output_map.width << std::endl;
std::cout << output_map.height << std::endl;
while (ros::ok())
{
output_cloud.header.stamp = ros::Time::now();
output_cloud.header.frame_id = "map";
publisher_cloud.publish(output_cloud);
output_map.header.stamp = ros::Time::now();
output_map.header.frame_id = "map";
publisher_map.publish(output_map);
ros::spinOnce();
loop_rate.sleep();
}
return (0);
}
```

Anexo 6. tfg_cloud.h

```
//===================================================================
//
// File: tfg_Cloud.h
//  Author: David Turbica, Ramiro Costa
//
//===================================================================

#ifndef __TFG_CLOUD_H_INCLUDED__
#define __TFG_CLOUD_H_INCLUDED__

#include <ros/ros.h>

//C++ includes
#include <string>
#include <vector>
#include <Eigen/Dense>

//ROS includes


//PCL includes
#include <pcl/io/pcd_io.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/filters/radius_outlier_removal.h>
#include <pcl/common/transforms.h>
#include <pcl/features/normal_3d_omp.h>

//TFG includes
#include "tfg_Logger.h"


class Cloud{
//Variables
public:
pcl::PointCloud<pcl::PointXYZ>::Ptr point_cloud;
pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals;

//Functions
public:
Cloud (void);
Cloud (pcl::PointCloud<pcl::PointXYZ>::Ptr new_point_cloud);
bool loadCloud (std::string path);
void saveCloud (std::string path);
void downsample (double resolution);
void removeOutliersStatisticaly (int neighbours = 50, double dev_const = 1.0);
void removeOutliersRadius (double radius = 0.1, int points = 10);
void applyTransform (Eigen::Matrix4f transformation_matrix);
void drawPoint (float x, float y, float z);
void computeNormals();
friend std::ostream & operator << (std::ostream &os, const Cloud &c){
os << "Cloud:  points[]: " << c.point_cloud->points.size () << std::endl
```

```
  << "        width: " << c.point_cloud->width << std::endl
  << "        height: " << c.point_cloud->height << std::endl
  << "        is_dense: " << c.point_cloud->is_dense << std::endl
  << "        sensor origin (xyz): ["
<< c.point_cloud->sensor_origin_.x () << ", "
<< c.point_cloud->sensor_origin_.y () << ", "
<< c.point_cloud->sensor_origin_.z () << "] / orientation (xyzw): ["
<< c.point_cloud->sensor_orientation_.x () << ", "
<< c.point_cloud->sensor_orientation_.y () << ", "
<< c.point_cloud->sensor_orientation_.z () << ", "
<< c.point_cloud->sensor_orientation_.w () << "]";
return os;
}
};

#endif //__TFG_CLOUD_H_INCLUDED
```

Anexo 7. tfg_Cloud.cpp

```
//===========================================================================
//
// File: tfg_Cloud.cpp
//  Author: David Turbica, Ramiro Costa
//
//===========================================================================

#include "tfg_processor/tfg_Cloud.h"

Cloud::Cloud (void){
//---------------------------------------------------------------------------
// Default constructor.
//---------------------------------------------------------------------------
this->point_cloud = pcl::PointCloud<pcl::PointXYZ>::Ptr (new pcl::PointCloud<pcl::PointXYZ>);
this->cloud_normals = pcl::PointCloud<pcl::PointNormal>::Ptr (new
pcl::PointCloud<pcl::PointNormal>);
//std::cout << "CLOUD instantiated" << std::endl;
}

Cloud::Cloud (pcl::PointCloud<pcl::PointXYZ>::Ptr new_point_cloud){
//---------------------------------------------------------------------------
// Constructor providing a PointCloud Pointer.
//---------------------------------------------------------------------------
this->point_cloud = new_point_cloud;
}

bool Cloud::loadCloud (std::string path){
//---------------------------------------------------------------------------
// It loads a PointCloud from a PCD file. It removes NaN points. Returns true
//  when PointCloud was loaded successfully, false when it was not.
//---------------------------------------------------------------------------
bool output = true;
if (pcl::io::loadPCDFile<pcl::PointXYZ> (path, *this->point_cloud) == -1 ||
pcl::io::loadPCDFile<pcl::PointNormal> (path, *this->cloud_normals) == -1){
output = false;
}
else{
std::vector<int> index1;
std::vector<int> index2;
pcl::removeNaNFromPointCloud (*this->point_cloud, *this->point_cloud, index1);
pcl::removeNaNFromPointCloud (*this->cloud_normals, *this->cloud_normals, index2);
}
return output;
}

void Cloud::saveCloud (std::string path){
//---------------------------------------------------------------------------
// Saves a PointCloud to a .pcd file.
//---------------------------------------------------------------------------
pcl::io::savePCDFile (path, *this->point_cloud);
}

void Cloud::downsample (double resolution){
//---------------------------------------------------------------------------
```

```
// Downsamples its PointCloud.
//------------------------------------------------------------------------
pcl::PointCloud<pcl::PointXYZ>::Ptr point_cloud_filtered (new pcl::PointCloud<pcl::PointXYZ>);
pcl::VoxelGrid<pcl::PointXYZ> filter;
filter.setInputCloud (this->point_cloud);
filter.setLeafSize (resolution, resolution, resolution);
filter.filter (*point_cloud_filtered);
this->point_cloud = point_cloud_filtered;

pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals_filtered (new
pcl::PointCloud<pcl::PointNormal>);
pcl::VoxelGrid<pcl::PointNormal> filter2;
filter2.setInputCloud (this->cloud_normals);
filter2.setLeafSize (resolution, resolution, resolution);
filter2.filter (*cloud_normals_filtered);
this->cloud_normals = cloud_normals_filtered;/**/
}

void Cloud::removeOutliersStatisticaly (int neighbours, double dev_const){
//------------------------------------------------------------------------
// Statistical filter to remove outliers
//------------------------------------------------------------------------
pcl::PointCloud<pcl::PointXYZ>::Ptr point_cloud_filtered (new pcl::PointCloud<pcl::PointXYZ>);
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> filter;
filter.setInputCloud (this->point_cloud);
filter.setMeanK (neighbours);
filter.setStddevMulThresh (dev_const);
filter.filter (*point_cloud_filtered);
this->point_cloud = point_cloud_filtered;
}

void Cloud::removeOutliersRadius (double radius, int points){
//------------------------------------------------------------------------
// Remove lonely outliers
//------------------------------------------------------------------------
pcl::PointCloud<pcl::PointXYZ>::Ptr point_cloud_filtered (new pcl::PointCloud<pcl::PointXYZ>);
pcl::RadiusOutlierRemoval<pcl::PointXYZ> filter;
    filter.setInputCloud (this->point_cloud);
    filter.setRadiusSearch (radius);
    filter.setMinNeighborsInRadius (points);
    filter.filter (*point_cloud_filtered);
    this->point_cloud = point_cloud_filtered;
}

void Cloud::applyTransform (Eigen::Matrix4f transformation_matrix){
//------------------------------------------------------------------------
// Applies a Transformation to its point_cloud.
//------------------------------------------------------------------------
pcl::transformPointCloud (*this->point_cloud, *this->point_cloud, transformation_matrix);
}

void Cloud::computeNormals(){
pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals (new pcl::PointCloud<pcl::PointNormal>);
pcl::NormalEstimation<pcl::PointNormal, pcl::PointNormal> ne;
ne.setInputCloud (this->cloud_normals);
pcl::search::KdTree<pcl::PointNormal>::Ptr tree(new pcl::search::KdTree<pcl::PointNormal> ());
    ne.setSearchMethod (tree);
```

```
    ne.setRadiusSearch (0.05);
    ne.compute (*cloud_normals);
std::cout << "NORMALS COMPUTED" << std::endl;
*this->cloud_normals = *cloud_normals;
pcl::copyPointCloud(*this->point_cloud, *this->cloud_normals);

}
void Cloud::drawPoint (float x, float y, float z){
this->point_cloud->push_back(pcl::PointXYZ(x,y,z));
}
```

Anexo 8. tfg_CloudMsg.h

```
//============================================================================
//
// File: tfg_CloudMsg.h
//  Author: David Turbica
//
//============================================================================

#ifndef __TFG_CLOUDMSG_H_INCLUDED__
#define __TFG_CLOUDMSG_H_INCLUDED__

#include <ros/ros.h>

//C++ includes
#include <string>
#include <ostream>

//ROS includes


//PCL includes


//TFG includes



class CloudMsg{
//Variables
private:
int time;
std::string path;

//Functions
public:
CloudMsg (void);
void setTime (int new_time);
void setPath (std::string new_path);
int getTime (void);
std::string getPath (void);
friend std::ostream & operator << (std::ostream &os, const CloudMsg &cm){
os << "Cloud:  Time:  " << cm.time << "\n"
   << "       Path:  " << cm.path;
return os;
}
};

#endif //__TFG_CLOUDMSG_H_INCLUDED
```

Anexo 9. tfg_CloudMsg.cpp

```
//===================================================================
//
// File: tfg_CloudMsg.cpp
//  Author: David Turbica
//
//===================================================================

#include "tfg_processor/tfg_CloudMsg.h"

CloudMsg::CloudMsg (void){
//-------------------------------------------------------------------
// Default constructor.
//-------------------------------------------------------------------
this->time = 0;
this->path = "";
}

void CloudMsg::setTime (int new_time){
//-------------------------------------------------------------------
// Sets 'time' value
//-------------------------------------------------------------------
this->time = new_time;
}

void CloudMsg::setPath (std::string new_path){
//-------------------------------------------------------------------
// Sest 'path' value
//-------------------------------------------------------------------
this->path = new_path;
}

int CloudMsg::getTime (void){
//-------------------------------------------------------------------
// Returns its 'time' value
//-------------------------------------------------------------------
return this->time;
}

std::string CloudMsg::getPath (void){
//-------------------------------------------------------------------
// Returns its 'path' value
//-------------------------------------------------------------------
return this->path;
}
```

Anexo 10. tfg_Drone.h

```
//============================================================================
//
// File: tfg_Drone.h
//  Author: David Turbica
//
//============================================================================

#ifndef __TFG_DRONE_H_INCLUDED__
#define __TFG_DRONE_H_INCLUDED__

#include <ros/ros.h>

//C++ includes
#include <string>
#include <ostream>
#include <vector>
#include <tf2/LinearMath/Matrix3x3.h>
#include <tf2/LinearMath/Transform.h>
#include <tf2/LinearMath/Vector3.h>

//ROS includes


//PCL includes


//TFG includes
#include "tfg_Map.h"


class Drone{
//Variables
private:
std::vector<tf2::Transform> locations;
std::vector<bool> checked;
tf2::Transform location;
tf2::Transform location_before;
tf2::Transform location_estimation;
//Functions
public:
Drone (void);
void setLocation (tf2::Transform loc);
void setLocationEstimation (tf2::Transform loc);
tf2::Transform getLocation (void);
tf2::Transform getLocationEstimation (void);
tf2::Vector3 getVelocityEstimation (float period);
void drawTrayectory (Map &map);
friend std::ostream & operator << (std::ostream &os, const Drone &d){
os << "Drone:  First Location:"
<< std::setw(20) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[0][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[0][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
```

```
d.locations.front().getBasis()[0][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getOrigin()[0]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[1][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[1][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[1][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getOrigin()[1]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[2][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[2][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getBasis()[2][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.locations.front().getOrigin()[2]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 1.0
<< std::endl
<< std::endl
<< "       Last Checked Location:"
<< std::setw(13) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[0][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[0][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[0][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getOrigin()[0]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[1][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[1][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[1][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getOrigin()[1]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[2][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[2][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getBasis()[2][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << d.location.getOrigin()[2]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 1.0
<< std::endl
<< std::endl
<< "       Location Estimation:"
<< std::setw(15) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[0][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[0][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[0][2]
```

```
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getOrigin()[0]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[1][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[1][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[1][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getOrigin()[1]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[2][0]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[2][1]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getBasis()[2][2]
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) <<
d.location_estimation.getOrigin()[2]
<< std::endl
<< std::setw(43) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 1.0;
return os;
}
};

#endif //__TFG_DRONE_H_INCLUDED
```

Anexo 11. tfg_Drone.cpp

```
//=================================================================================
//
// File: tfg_Drone.cpp
//  Author:
//
//=================================================================================

#include "tfg_processor/tfg_Drone.h"

Drone::Drone (void){
//---------------------------------------------------------------------------------
// Default constructor.
//---------------------------------------------------------------------------------
}

void Drone::setLocation (tf2::Transform loc){
//---------------------------------------------------------------------------------
// Sets a new location. Adds that new location to the trayectory.
//---------------------------------------------------------------------------------
this->location_before = this->location;
this->location = loc;
this->locations.push_back (loc);
this->checked.push_back (true);
}

void Drone::setLocationEstimation (tf2::Transform loc){
//---------------------------------------------------------------------------------
// Sets a new location estimation. Adds that new location to the trayectory.
//---------------------------------------------------------------------------------
this->location_estimation = loc;
this->locations.push_back (loc);
this->checked.push_back (false);
}

tf2::Transform Drone::getLocation (void){
//---------------------------------------------------------------------------------
// Returns its 'location' value.
//---------------------------------------------------------------------------------
return this->location;
}

tf2::Transform Drone::getLocationEstimation (void){
//---------------------------------------------------------------------------------
// Returns its 'location_estimation' value.
//---------------------------------------------------------------------------------
return this->location_estimation;
}

tf2::Vector3 Drone::getVelocityEstimation (float period){
//---------------------------------------------------------------------------------
// It computes the to latest locations checked to estimate velocity.
//---------------------------------------------------------------------------------
return (this->location.getOrigin() - this->location_before.getOrigin()) / period;
}
```

```
void Drone::drawTrayectory (Map &map){
//-------------------------------------------------------------------------
// It draws drone trayectory points in its cloud.
//-------------------------------------------------------------------------
std::vector<tf2::Transform>::iterator it_loc = this->locations.begin();
std::vector<bool>::iterator it_che = this->checked.begin();
while (it_loc != this->locations.end() && it_che != this->checked.end()){
map.drawLocation (*it_loc, *it_che);
it_loc ++;
it_che ++;
}
}
```

Anexo 12. tfg_Filter.h

```
//============================================================================
//
// File: tfg_Filter.h
//  Author: David Turbica
//
//============================================================================

#ifndef __TFG_FILTER_H_INCLUDED__
#define __TFG_FILTER_H_INCLUDED__

class Filter{
//Variables
public:
double measures [50];
int number_of_measures;
int iterator;
bool full;
//Functions
public:
Filter (void);
Filter (int new_number_of_measures);
void avoidStartAttenuance (void);
void addMeasure (double new_measure);
double calcMean (void);
double calcMedian (void);
};

#endif //__TFG_FILTER_H_INCLUDED
```

Anexo 13. tfg_Filter.cpp

```
//=========================================================================
//
// File: tfg_Filter.cpp
//  Author: David Turbica
//
//=========================================================================

#include "tfg_processor/tfg_Filter.h"

Filter::Filter (void){
//-------------------------------------------------------------------------
// Default constructor
//-------------------------------------------------------------------------
this->number_of_measures = 0;
this->iterator = 0;
this->full = true;
}

Filter::Filter (int new_number_of_measures){
//-------------------------------------------------------------------------
// Constructor given the lenght of the filter, the number of measures to be
//  processed.
//-------------------------------------------------------------------------
this->number_of_measures = new_number_of_measures;
this->iterator = 0;
this->full = true;
for(int i=0; i<this->number_of_measures; i++){
this->measures[i] = 0.0;
}
}

void Filter::avoidStartAttenuance (void){
//-------------------------------------------------------------------------
// It will make the filter only compute measures that had already been added,
//  preventing the filter to compute as zero all the measures not already set.
//-------------------------------------------------------------------------
this->full = false;
}

void Filter::addMeasure (double new_measure){
//-------------------------------------------------------------------------
// It adds cyclicly a new measure to the filter. FIFO stack.
//-------------------------------------------------------------------------
this->measures[iterator] = new_measure;
this->iterator = (this->iterator + 1) % this->number_of_measures;
if (!this->full) if (iterator == 0) this->full = true;
}

double Filter::calcMean (void){
//-------------------------------------------------------------------------
// It returns the mean of all the measures contained in the filter, even
//  not already set measures which will be computed as cero. Unless
//  'avoidStartAttenuance ()' had been called, where only added measures
//  will be computed.
```

```
//---------------------------------------------------------------------------
double result = 0.0;
int last_measure = this->number_of_measures;
if (!this->full) last_measure = this->iterator;
for(int i=0; i<last_measure; i++){
result += this->measures[i];
}
result = result / last_measure;
return result;
}

double Filter::calcMedian (void){
//---------------------------------------------------------------------------
// It returns the median of all the measures contained in the filter, even
//  not already set measures which will be computed as cero. Unless
//  'avoidStartAttenuance ()' had been called, where only added measures
//  will be computed.
//---------------------------------------------------------------------------
double result = 0.0;
int last_measure = this->number_of_measures;
if (!this->full) last_measure = this->iterator;
if(last_measure > 0){
double sorted_measures[50];
int number_of_sorted_measures = 1;
sorted_measures[0] = this->measures[0];
for(int i=1; i<last_measure; i++){
bool found = false;
for(int e=number_of_sorted_measures-1; e>=0 && !found; e--){
if(this->measures[i]>sorted_measures[e]){
sorted_measures[e+1] = sorted_measures[e];
}
else{
sorted_measures[e+1] = this->measures[i];
found = true;
}
}
if(!found){
sorted_measures[0] = this->measures[i];
}
number_of_sorted_measures ++;
}
result = sorted_measures[number_of_sorted_measures/2];
}
return result;
}
```

Anexo 14. tfg_FilterVector3.h

```
//=================================================================
//
// File: tfg_FilterVector3.h
//  Author: David Turbica
//
//=================================================================

#ifndef __TFG_FILTERVECTOR3_H_INCLUDED__
#define __TFG_FILTERVECTOR3_H_INCLUDED__

//C++ includes
#include <tf2/LinearMath/Vector3.h>

//ROS includes


//PCL includes


//TFG includes
#include "tfg_Filter.h"


class FilterVector3{
//Variables
public:
Filter filter_x;
Filter filter_y;
Filter filter_z;
//Functions
public:
FilterVector3 (void);
FilterVector3 (int new_number_of_measures);
void avoidStartAttenuance (void);
void addMeasure (tf2::Vector3 new_measure);
tf2::Vector3 calcMean (void);
tf2::Vector3 calcMedian (void);
};

#endif //__TFG_FILTERVECTOR3_H_INCLUDED
```

Anexo 15. tfg_FilterVector3.cpp

```
//=========================================================================
//
// File: tfg_FilterVector3.cpp
//  Author: David Turbica
//
//=========================================================================

#include "tfg_processor/tfg_FilterVector3.h"

FilterVector3::FilterVector3 (void){
//-------------------------------------------------------------------------
// Default constructor.
//-------------------------------------------------------------------------
this->filter_x = Filter();
this->filter_y = Filter();
this->filter_z = Filter();
}

FilterVector3::FilterVector3 (int new_number_of_measures){
//-------------------------------------------------------------------------
// Constructor given the lenght of the filter, the number of meassures to be
//  processed.
//-------------------------------------------------------------------------
this->filter_x = Filter (new_number_of_measures);
this->filter_y = Filter (new_number_of_measures);
this->filter_z = Filter (new_number_of_measures);
}

void FilterVector3::avoidStartAttenuance (void){
//-------------------------------------------------------------------------
// It will make the filter only compute measures that had already been added,
//  preventing the filter to compute as zero all the measures not already set.
//-------------------------------------------------------------------------
this->filter_x.avoidStartAttenuance();
this->filter_y.avoidStartAttenuance();
this->filter_z.avoidStartAttenuance();
}

void FilterVector3::addMeasure (tf2::Vector3 new_measure){
//-------------------------------------------------------------------------
// It adds cyclicly a new meassure to the filter. FIFO stack.
//-------------------------------------------------------------------------
this->filter_x.addMeasure (new_measure.getX());
this->filter_y.addMeasure (new_measure.getY());
this->filter_z.addMeasure (new_measure.getZ());
}

tf2::Vector3 FilterVector3::calcMean (void){
//-------------------------------------------------------------------------
// It returns the mean of all the meassures contained in the filter, even not
//  already set meassures which will be computed as vectors (0,0,0).
//-------------------------------------------------------------------------
tf2::Vector3 output;
output.setX(this->filter_x.calcMean());
```

```
output.setY(this->filter_y.calcMean());
output.setZ(this->filter_z.calcMean());
return output;
}

tf2::Vector3 FilterVector3::calcMedian (void){
//-----------------------------------------------------------------------------
// It returns the median of all the meassures contained in the filter, even
//  not already set meassures which will be computed as vectors (0,0,0).
//-----------------------------------------------------------------------------
tf2::Vector3 output;
output.setX(this->filter_x.calcMedian());
output.setY(this->filter_y.calcMedian());
output.setZ(this->filter_z.calcMedian());
return output;
}
```

Anexo 16. tfg_Logger.h

```
//============================================================================
//
// File: tfg_FilterVector3.h
//  Author: David Turbica
//
//============================================================================

#ifndef __TFG_FILTERVECTOR3_H_INCLUDED__
#define __TFG_FILTERVECTOR3_H_INCLUDED__

//C++ includes
#include <tf2/LinearMath/Vector3.h>

//ROS includes


//PCL includes


//TFG includes
#include "tfg_Filter.h"


class FilterVector3{
//Variables
public:
Filter filter_x;
Filter filter_y;
Filter filter_z;
//Functions
public:
FilterVector3 (void);
FilterVector3 (int new_number_of_measures);
void avoidStartAttenuance (void);
void addMeasure (tf2::Vector3 new_measure);
tf2::Vector3 calcMean (void);
tf2::Vector3 calcMedian (void);
};

#endif //__TFG_FILTERVECTOR3_H_INCLUDED
```

Anexo 17. Tfg_Map.h

```
//================================================================
//
// File: tfg_Map.h
//  Author: David Turbica, Ramiro Costa
//
//================================================================

#ifndef __TFG_MAP_H_INCLUDED__
#define __TFG_MAP_H_INCLUDED__

#include <ros/ros.h>

//C++ includes
#include <string>
#include <Eigen/Dense>
#include <tf2/LinearMath/Transform.h>
#include <tf2/LinearMath/Vector3.h>
#include <tf2/LinearMath/Matrix3x3.h>

//ROS includes


//PCL includes
#include <pcl/registration/icp.h>
#include <pcl/common/transforms.h>
#include <pcl/registration/transformation_estimation_lm.h>
#include <pcl/registration/transformation_estimation_2D.h>
#include <pcl/registration/correspondence_rejection_trimmed.h>
#include <pcl/registration/correspondence_estimation.h>
#include <pcl/registration/transformation_estimation_point_to_plane.h>
#include <pcl/registration/transformation_estimation_point_to_plane_lls.h>
#include <pcl/registration/transformation_estimation.h>
#include <pcl/features/normal_3d_omp.h>



//TFG includes
#include "tfg_Cloud.h"
#include "tfg_Logger.h"
#include "tfg_utilities.h"

class Map{
//Variables
public:
Cloud cloud;
Cloud output_cloud;
bool p2plane;
std::vector<float> error;

//Functions
public:
Map (void);
bool loadMap (std::string path);
```

```
void saveMap (std::string path);
void downsample (double resolution);
void removeOutliersStatisticaly (int neighbours = 50, double dev_const = 1.0);
void removeOutliersRadius (double radius = 0.1, int points = 10);
void applyTransform (Eigen::Matrix4f transformation_matrix);
tf2::Transform locate (Cloud frame, tf2::Transform estimation);
void drawLocation (tf2::Transform loc, bool type);
void copyPC ();
void saveError();
friend std::ostream & operator << (std::ostream &os, const Map &m){
os << "Map:    Cloud:  points[]: " << m.cloud.point_cloud->points.size () << std::endl
   << "               width: " << m.cloud.point_cloud->width << std::endl
   << "               height: " << m.cloud.point_cloud->height << std::endl
   << "               is_dense: " << m.cloud.point_cloud->is_dense << std::endl
   << "               sensor origin (xyz): ["
<< m.cloud.point_cloud->sensor_origin_.x () << ", "
<< m.cloud.point_cloud->sensor_origin_.y () << ", "
<< m.cloud.point_cloud->sensor_origin_.z () << "] / orientation (xyzw): ["
<< m.cloud.point_cloud->sensor_orientation_.x () << ", "
<< m.cloud.point_cloud->sensor_orientation_.y () << ", "
<< m.cloud.point_cloud->sensor_orientation_.z () << ", "
<< m.cloud.point_cloud->sensor_orientation_.w () << "]";
return os;
}
};

#endif //__TFG_MAP_H_INCLUDED
```

Anexo 18. tfg_Map.cpp

```
//===================================================================
//
// File: tfg_Map.cpp
//  Author: David Turbica, Ramiro Costa
//
//===================================================================

#include "tfg_processor/tfg_Map.h"


Map::Map (void){
//-------------------------------------------------------------------
// Default constructor.
//-------------------------------------------------------------------
p2plane = false;
}

bool Map::loadMap (std::string path){
//-------------------------------------------------------------------
// Loads a PointCloud from a .pcd file. Returns true when PointCloud was
//  loaded successfully, false when it was not.
//-------------------------------------------------------------------
return this->cloud.loadCloud (path);
}

void Map::saveMap (std::string path){
//-------------------------------------------------------------------
// Saves a PointCloud to a .pcd file.
//-------------------------------------------------------------------
this->cloud.saveCloud (path);
}

void Map::downsample (double resolution) {
//-------------------------------------------------------------------
// Downsamples its Cloud to the resolution provided.
//-------------------------------------------------------------------
this->cloud.downsample (resolution);
}

void Map::removeOutliersStatisticaly (int neighbours, double dev_const){
//-------------------------------------------------------------------
// Statistical filter to remove outliers
//-------------------------------------------------------------------
this->cloud.removeOutliersStatisticaly (neighbours, dev_const);
}

void Map::removeOutliersRadius (double radius, int points){
//-------------------------------------------------------------------
// Remove lonely outliers
//-------------------------------------------------------------------
this->cloud.removeOutliersRadius (radius, points);
}

void Map::applyTransform (Eigen::Matrix4f transformation_matrix){
```

```
//---------------------------------------------------------------------------
// Applies a Transformation to its point_cloud.
//---------------------------------------------------------------------------
this->cloud.applyTransform (transformation_matrix);
}

tf2::Transform Map::locate (Cloud frame, tf2::Transform estimation){
//---------------------------------------------------------------------------
// Locates a Cloud in map. Location estimation is used if 'estimation' is
//  provided. It returns the location of the cloud in map.
//---------------------------------------------------------------------------
//Convert estimation form tf2::Transform to Eigen:Matrix4d


Eigen::Matrix4f eigen_estimation = TransformToEigenMatrix (estimation);
pcl::registration::TransformationEstimationPointToPlaneLLS <pcl::PointNormal,
pcl::PointNormal, float>::Ptr tep2plls (new
pcl::registration::TransformationEstimationPointToPlaneLLS <pcl::PointNormal,
pcl::PointNormal, float> ());

Eigen::Matrix4f transformation_matrix = Eigen::Matrix4f::Identity ();
pcl::PointCloud<pcl::PointNormal> output_n;
pcl::PointCloud<pcl::PointXYZ> output;
pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
pcl::IterativeClosestPoint<pcl::PointNormal, pcl::PointNormal> icp_n;

std::cout << "1" << std::endl;
if(!p2plane){
//icp.setMaxCorrespondenceDistance(1);

//icp_n.setEuclideanFitnessEpsilon(1.2);
//icp_n.setTransformationEpsilon (1.2);

//icp_n.setMaxCorrespondenceDistance(1);
//icp_n.setTransformationEpsilon(0.5);
icp.setMaximumIterations(50);

icp.setInputSource (frame.point_cloud); //CLOUD
icp.setInputTarget (this->cloud.point_cloud); //MAP

icp.align (output, eigen_estimation);
std::cout << "2" << std::endl;


if (icp.hasConverged () && icp.getFitnessScore () <= 0.001){
std::cout << "\nICP has converged, score is " << icp.getFitnessScore () << std::endl;
std::cout << "\nICP transformation " << icp.nr_iterations_ << " : cloud_icp -> cloud_in" <<
std::endl;
this->error.push_back(icp.getFitnessScore());
transformation_matrix = icp.getFinalTransformation();
printEigenMatrix4f (transformation_matrix);

*(this->output_cloud.point_cloud) = output;
return EigenMatrixToTransform (transformation_matrix);

}else if(icp.getFitnessScore () > 0.001){
p2plane = true;
}
}
if(p2plane){
```

```
icp_n.setInputSource (frame.cloud_normals); //CLOUD
icp_n.setInputTarget (this->cloud.cloud_normals); //MAP
icp_n.setMaximumIterations (50);
//icp_n.setMaxCorrespondenceDistance(1);
//icp_n.setTransformationEpsilon(0.5);
//icp_n.setEuclideanFitnessEpsilon(1.2);
//icp_n.setTransformationEpsilon (1.2);
icp_n.setTransformationEstimation(tep2plls);
icp_n.align (output_n, eigen_estimation);
if(icp_n.hasConverged() && icp_n.getFitnessScore () <= 0.003){
std::cout << "\nICP has converged, score is " << icp_n.getFitnessScore () << std::endl;
std::cout << "\nICP transformation " << icp_n.nr_iterations_ << " : cloud_icp -> cloud_in" <<
std::endl;
this->error.push_back(icp_n.getFitnessScore());
transformation_matrix = icp_n.getFinalTransformation();
printEigenMatrix4f (transformation_matrix);
*(this->output_cloud.cloud_normals) = output_n;
this->copyPC();
return EigenMatrixToTransform (transformation_matrix);
}else {
p2plane = false;
this->error.push_back(icp_n.getFitnessScore());
}
}
PCL_ERROR ("\nICP has not converged.\n");
return estimation;
}

void Map::drawLocation (tf2::Transform loc, bool type){
this->cloud.drawPoint (loc.getOrigin()[0], loc.getOrigin()[1], loc.getOrigin()[2]);
}

void Map::copyPC ()
{
this->output_cloud.point_cloud->clear();
   //std::cout << "cloud out  " << cloud_out << std::endl;
this->output_cloud.point_cloud->width    = this->output_cloud.cloud_normals->width;
//std::cout << "cloud width  " << cloud_out->width << std::endl;
this->output_cloud.point_cloud->height   = this->output_cloud.cloud_normals->height;
//std::cout << "cloud height  " << cloud_out->height << std::endl;
this->output_cloud.point_cloud->is_dense = this->output_cloud.cloud_normals->is_dense;
//std::cout << "cloud is_dense  " << cloud_out->is_dense << std::endl;
int i=0;

while (i < this->output_cloud.cloud_normals->points.size ()){
//std::cout << i << std::endl;
pcl::PointXYZ point (this->output_cloud.cloud_normals->points[i].x, this-
>output_cloud.cloud_normals->points[i].y, this->output_cloud.cloud_normals->points[i].z);
this->output_cloud.point_cloud->push_back(point);
//std::cout << point.x << "," << point.y << "," << point.z << std::endl;
//std::cout << this->cloud_normals->points[i].x << "  " << this->cloud_normals->points[i].y << "  "
<< this->cloud_normals->points[i].z << std::endl;
++i;
}
}
void Map::saveError(){
std::ofstream file;
```

```
file.open("src/tfg/error.txt");

for(int i=0;i< this->error.size();i++){
file << this->error[i] << "\n";
}
file.close();
}
```

Anexo 19. tfg_Process.h

```
//================================================================
//
// File: tfg_Process.h
//  Author: David Turbica
//
//================================================================

#ifndef __TFG_PROCESS_H_INCLUDED__
#define __TFG_PROCESS_H_INCLUDED__


//C++ includes


//ROS includes


//PCL includes


//TFG includes
#include "tfg_Timer.h"


namespace tfg {


class Process{
//Variables
public:
int steps;
int current_step;
float average_time;
double last_time;
Timer timer;

//Functions
public:
Process (void);
void create (int n_steps);
void start (void);
double stop (void);
double step (void);
double lastTime (void);
int progress (void);
float timeRemainingEstimation (void);
};


} //end of namespace tfg

#endif //__TFG_PROCESS_H_INCLUDED
```

Anexo 20. tfg_Process.cpp

```
//===================================================================
//
// File: tfg_Process.cpp
//  Author: David Turbica
//
//===================================================================

#include "tfg_processor/tfg_Process.h"

namespace tfg {

Process::Process (void){
//-------------------------------------------------------------------
// Default constructor.
//-------------------------------------------------------------------
this->steps = 0;
this->current_step = 0;
this->average_time = 0.0;
this->last_time = 0.0;
}

void Process::create (int n_steps){
//-------------------------------------------------------------------
// Sets 'steps' value.
//-------------------------------------------------------------------
this->steps = n_steps;
this->current_step = 0;
this->average_time = 0.0;
this->last_time = 0.0;
}

void Process::start (void){
//-------------------------------------------------------------------
// Starts meassuring time.
//-------------------------------------------------------------------
this->timer.start();
}

double Process::stop (void){
//-------------------------------------------------------------------
// Returns the time elapsed since 'start()' was called.
//-------------------------------------------------------------------
this->last_time = this->timer.stop();
return this->last_time;
}

double Process::step (void){
//-------------------------------------------------------------------
// Advances one step. Returns the time elapsed sice last 'start()' was call.
//  Also recalculates the period time average.
//-------------------------------------------------------------------
double period = this->timer.period ();
this->last_time = this->timer.step();
this->current_step ++;
```

```
this->average_time += ((period - this->average_time) / this->current_step);
return this->last_time;
}
double Process::lastTime (void){
//---------------------------------------------------------------------------
// Returns last_time
//---------------------------------------------------------------------------
return this->last_time;
}

int Process::progress (void){
//---------------------------------------------------------------------------
// Returns the progress (%) of the process.
//---------------------------------------------------------------------------
return (int)((100 * (long)this->current_step) / (long)this->steps);
}

float Process::timeRemainingEstimation (void){
//---------------------------------------------------------------------------
// Returns the estimated reminding time for the process to be completed.
//---------------------------------------------------------------------------
return (this->steps - this->current_step) * this->average_time;
}

} //end of namespace tfg
```

Anexo 21. tfg_processor.h

```
//=================================================================
//
// File: tfg_processor.h
//  Author: David Turbica, Ramiro Costa
//
//=================================================================

#ifndef __TFG_PROCESSOR_H_INCLUDED__
#define __TFG_PROCESSOR_H_INCLUDED__

#include <ros/ros.h>
#include <signal.h>

//C++ includes
#include <string>
#include <iostream>
#include <fstream>
#include <limits>
#include <stdlib.h>
#include <Eigen/Dense>
#include <tf2/LinearMath/Vector3.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/LinearMath/Transform.h>


//ROS includes
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/Imu.h>
#include <pcl_conversions/pcl_conversions.h>
#include <geometry_msgs/Transform.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PoseArray.h>
#include <geometry_msgs/Pose.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <visualization_msgs/Marker.h>
#include <visualization_msgs/MarkerArray.h>

//PCL includes
#include "pcl_ros/point_cloud.h"
#include <pcl/features/normal_3d.h>


//TFG includes
#include "tfg_CloudMsg.h"
#include "tfg_ImuMsg.h"
#include "tfg_Map.h"
#include "tfg_Cloud.h"
#include "tfg_Drone.h"
#include "tfg_Imu.h"
#include "tfg_FilterVector3.h"
#include "tfg_Process.h"
#include "tfg_Timer.h"
#include "tfg_Logger.h"
```

```
//Command parser variables
std::string main_path;
std::string sequence;
int filter_size;

//Global Variables
bool sigint_called = false;
bool more_data;
long simulation_time; //in miliseconds
std::string path_clds;
std::string path_clds_data;
std::string path_imus_data;
std::string path_parameter;
std::string path_map;
std::string path_origin;
std::string path_imu_calibration;
std::string path_imu_calibration_params;
Map map;
Drone drone;
Imu imu;

//Variables CAM
int cld_counter;
bool next_cld;
CloudMsg cld_msg;
int time_last_cloud;

//Variables IMU
int imu_counter;
int imu_counter_calibration;
bool next_imu;
ImuMsg imu_msg;
int filter_counter;
FilterVector3 filter;

//Debugging step-by-step
bool stop_cld = false;
bool stop_imu = true;

//Files
std::ifstream origin_file;
std::ifstream cld_file;
std::ifstream imu_file;
std::ifstream param_file;
std::ifstream calib_file;
std::ifstream calib_params_file;

//Functions
int main (int argc, char** argv);
void customSigintHandler(int sig);
void printHelp (char* program_name);
void pauseSimulation (void);

#endif //__TFG_PROCESSOR_H_INCLUDED
```

Anexo 22. tfg_processor.cpp

```
//=================================================================================
//
// File: tfg_processor.cpp
//  Author: David Turbica, Ramiro Costa
//
//=================================================================================

#include "tfg_processor/tfg_processor.h"


int main (int argc, char** argv) {
//---------------------------------------------------------------------------
// Main function. It initializes the whole program and sets the timers and
//  and topics' subscribers and publishers.
//---------------------------------------------------------------------------

//=================================================================================
//  Default arguments
    //=============================================================================
==
main_path = "src/tfg/tfg_processor/map";
sequence = "2";
filter_size = 20;
LOG_LEVEL = 1; //Info level
stop_cld = false;
//stop_imu = false;

//=================================================================================
//  Parse arguments
    //=============================================================================
==
for (int i=1; i<argc; i++){
std::string arg (argv[i]);
if (arg == "-h" || arg == "--help"){
printHelp(argv[0]);
return 0;
}

else if (arg == "-d" || arg == "--debug"){
LOG_LEVEL = 0;
}

else if (arg == "-i" || arg == "--info"){
LOG_LEVEL = 1;
}

else if (arg == "-e" || arg == "--error"){
LOG_LEVEL = 2;
}

else if (arg == "-n" || arg == "--nolog"){
LOG_LEVEL = 3;
}
```

```
else if (arg == "-di" || arg == "--debugimu"){
LOG_LEVEL = 0;
stop_cld = false;
//stop_imu = true;
}

else if (arg == "-dc" || arg == "--debugcloud"){
LOG_LEVEL = 0;
stop_cld = true;
//stop_imu = false;
}

else if (arg == "-db" || arg == "--debugboth"){
LOG_LEVEL = 0;
stop_cld = true;
//stop_imu = true;
}

else if (arg == "-m" || arg == "--map"){
i++;
if (i < argc){
main_path = std::string (argv[i]);
}
else{
printHelp(argv[0]);
return 0;
}
}

else if (arg == "-s" || arg == "--seq"){
i++;
if (i < argc){
sequence = std::string (argv[i]);
}
else{
printHelp(argv[0]);
return 0;
}
}

else if (arg == "-f" || arg == "--filter"){
i++;
if (i < argc){
filter_size = std::atoi (argv[i]);
}
else{
printHelp(argv[0]);
return 0;
}
}

else{
printHelp(argv[0]);
return 0;
}
}
```

```
//=============================================================================
//  Print header
    //=============================================================================
==
PRINT_INFO
("=============================================================================");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("|                        TFG_PROCESSOR                              |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|===================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Author:       David Turbica, Ramiro Costa                         |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Description:      ICP-based RGBd camera location system on a      |");
PRINT_INFO ("|                   known environment                               |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("=============================================================================");

//=============================================================================
//  Initialization
    //=============================================================================
==

//Initialize ROS
ros::init (argc, argv, "tfg_processor");
signal(SIGINT, customSigintHandler);
ros::NodeHandle n;

//Suppresses any PCL's output to the console.
//pcl::console::setVerbosityLevel(pcl::console::L_ALWAYS);

//Initialize global variables
more_data = true;
simulation_time = 0;
path_clds = main_path + "/sequence_" + sequence + "/clouds/";
path_clds_data = main_path + "/sequence_" + sequence + "/clouds_data.txt";
path_parameter = main_path + "/sequence_" + sequence + "/parameters.txt";
path_map = main_path + "/mapa_inverso_cuarto.pcd";
path_origin = main_path + "/sequence_" + sequence + "/origin.txt";

PRINT_INFO ("");
PRINT_INFO ("-----------------------------------------------------------------");
PRINT_INFO ("Loading data from the following sources:");
PRINT_INFO ("    map.pcd                  " << path_map);
PRINT_INFO ("    origin.txt               " << path_origin);
PRINT_INFO ("    clouds/                  " << path_clds);
PRINT_INFO ("    clouds_data.txt          " << path_clds_data);
PRINT_INFO ("    imus_data.txt            " << path_imus_data);
PRINT_INFO ("    imu_calibration_params.txt   " << path_imu_calibration_params);
PRINT_INFO ("    imu_calibration.txt      " << path_imu_calibration);
PRINT_INFO ("    parameters.txt           " << path_parameter);
PRINT_INFO ("-----------------------------------------------------------------");
PRINT_INFO ("Checking files:");

//=============================================================================
```

```
//  Load map from 'map.pcd' file
    //=============================================================================
==

if (map.loadMap (path_map)){
PRINT_INFO ("    mapa.pcd                    LOADED");

//Downsample 0.025
map.downsample(0.04);
map.cloud.computeNormals();

//Remove outliers
//map.removeOutliersStatisticaly (50, 3);
//map.removeOutliersRadius (0.06, 4);
}else{
PRINT_ERROR ("    map.pcd                     ERROR");
PRINT_INFO ("-------------------------------------------------------------------");
PRINT_INFO ("");
PRINT_ERROR ("Unable to load map from '" << path_map << "'");
PRINT_INFO ("");
PRINT_INFO
("=====================================================================");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("|                      TFG_PROCESSOR                                |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|==================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Author:              David Turbica, Ramiro Costa                  |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Description:      ICP-based RGBd camera location system on a      |");
PRINT_INFO ("|                   known environment                               |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|==================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Node terminated:  No map to work on.                              |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("=====================================================================");
ros::shutdown();
return 0;
}

//=============================================================================
//  Parse 'origin.txt' file
    //=============================================================================
==
origin_file.open (path_origin.c_str());
if (origin_file.is_open()){
PRINT_INFO ("    origin.txt                  LOADED");
tf2::Matrix3x3 rotation;
tf2::Vector3 position;
//Parse the file
float cell;
origin_file >> cell;
rotation[0][0] = cell;
```

```
origin_file >> cell;
rotation[0][1] = cell;
origin_file >> cell;
rotation[0][2] = cell;
origin_file >> cell;
position[0] = cell;
origin_file >> cell;
rotation[1][0] = cell;
origin_file >> cell;
rotation[1][1] = cell;
origin_file >> cell;
rotation[1][2] = cell;
origin_file >> cell;
position[1] = cell;
origin_file >> cell;
rotation[2][0] = cell;
origin_file >> cell;
rotation[2][1] = cell;
origin_file >> cell;
rotation[2][2] = cell;
origin_file >> cell;
position[2] = cell;
//Close the file
origin_file.close();
//Initialize DRONE location
tf2::Transform origin_location (rotation, position);
drone.setLocation (origin_location);
PRINT_INFO(PRINT_MATRIX(origin_location.getBasis()));
PRINT_INFO(*origin_location.getOrigin());
}else{
PRINT_ERROR ("    origin.txt                ERROR");
PRINT_INFO ("-------------------------------------------------------------------");
PRINT_INFO ("");
PRINT_ERROR ("Unable to load origin location from '" << path_origin << "'");
PRINT_INFO ("");
PRINT_INFO
("=====================================================================");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("|                    TFG_PROCESSOR                                  |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|==================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Author:            David Turbica, Ramiro Costa                    |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Description:     ICP-based RGBd camera location system on a       |");
PRINT_INFO ("|                  known environment                                |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|==================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Node terminated:  No origin file.                                 |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("=====================================================================");
ros::shutdown();
return 0;
```

```
}

//=========================================================================
//  Initialize CAM variables
    //=========================================================================
==
cld_file.open (path_clds_data.c_str());
cld_counter = 0;
next_cld = true;
time_last_cloud = 0;

//=========================================================================
//  Checking files 'cld_file'
    //=========================================================================
==
if (cld_file.is_open()){
PRINT_INFO ("    clouds_data.txt             LOADED");
}else{
PRINT_ERROR ("    clouds_data.txt             ERROR");
PRINT_INFO ("-------------------------------------------------------------------");
PRINT_INFO ("");
PRINT_ERROR ("Unable to load cloud info from '" << path_clds_data << "'");
PRINT_INFO ("");
PRINT_INFO
("=====================================================================");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("|                    TFG_PROCESSOR                                  |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|===================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Author:             David Turbica, Ramiro Costa                   |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Description:      ICP-based RGBd camera location system on a      |");
PRINT_INFO ("|                   known environment                               |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("|===================================================================|");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("| Node terminated:  No cloud data to work with.                     |");
PRINT_INFO ("|                                                                   |");
PRINT_INFO
("=====================================================================");
ros::shutdown();
return 0;
}


//=========================================================================
//  Parse 'parameters.txt' file
    //=========================================================================
==
param_file.open (path_parameter.c_str());
if (param_file.is_open()){
PRINT_INFO ("    parameters.txt              LOADED");
PRINT_INFO ("-------------------------------------------------------------------");
```

```
std::string counter_str;
//Getting number of clouds to process
std::getline (param_file, counter_str, ' ');
std::getline (param_file, counter_str, '\n');
cld_counter = std::atoi (counter_str.c_str());
//Closing file
param_file.close();
}else{
PRINT_ERROR ("    parameters.txt              ERROR");
PRINT_INFO ("--------------------------------------------------------------------");
PRINT_INFO ("");
PRINT_ERROR ("File 'parameters.txt' not found");
PRINT_INFO ("");
PRINT_INFO
("======================================================================");
PRINT_INFO ("|                                                                    |");
PRINT_INFO ("|                      TFG_PROCESSOR                                 |");
PRINT_INFO ("|                                                                    |");
PRINT_INFO
("|====================================================================|");
PRINT_INFO ("|                                                                    |");
PRINT_INFO ("| Author:              David Turbica, Ramiro Costa                   |");
PRINT_INFO ("|                                                                    |");
PRINT_INFO ("| Description:      ICP-based RGBd camera location system on a       |");
PRINT_INFO ("|                   known environment                                |");
PRINT_INFO ("|                                                                    |");
PRINT_INFO
("|====================================================================|");
PRINT_INFO ("|                                                                    |");
PRINT_INFO ("| Node terminated:  No parameters for PointCloud and Imu to work     |");
PRINT_INFO ("|                   with.                                            |");
PRINT_INFO ("|                                                                    |");
PRINT_INFO
("======================================================================");
ros::shutdown();
return 0;
}

//**********************************************************************
// TEST ICP END
//**********************************************************************


int cld_counter_BU = cld_counter;
double cld_min = 1000000000.0;
double cld_max = 0.0;

//=============================================================================
//  Print some debug info
   //=============================================================================
==
PRINT_DEBUG (map);
PRINT_DEBUG ("--------------------------------------------------------------");
PRINT_DEBUG (drone);
PRINT_DEBUG ("--------------------------------------------------------------");
PRINT_DEBUG ("Number of clouds to process:  " << cld_counter);
PRINT_DEBUG ("--------------------------------------------------------------");
```

```
PRINT_DEBUG ("------------------------------------------------------------------------");

//=============================================================================
//  Simulation progress control
    //=============================================================================
==
PRINT_INFO ("Processing sequence:");
PRINT_INFO ("     Progress    Sim.Time    Pro.Time    Time.Remaining");
tfg::Timer cld_timer;
tfg::Process process;
process.create (cld_counter);
process.start();

//-----------------------------------------------------------------------
//  Prepare variables for map and frame streaming
//-----------------------------------------------------------------------

sensor_msgs::PointCloud2 output_map;
sensor_msgs::PointCloud2 output_cloud;
pcl::toROSMsg(*(map.cloud.point_cloud), output_map);

ros::Publisher publisher_map = n.advertise<sensor_msgs::PointCloud2>("output_map", 10000);
ros::Publisher publisher_cloud = n.advertise<sensor_msgs::PointCloud2>("output_cloud", 10000);
ros::Publisher pub_transform = n.advertise<geometry_msgs::PoseStamped>("pose_transform", 1);
ros::Publisher marker_pub = n.advertise<visualization_msgs::Marker>("visualization_marker", 10);
ros::Publisher marker_pub2 = n.advertise<visualization_msgs::Marker>("visualization_marker", 10);
uint32_t shape = visualization_msgs::Marker::ARROW;

tf2::Vector3 last_position;
float previous_timer = 1.0;
Eigen::Matrix4f estimation;
/*
//sequence 1
estimation     <<   -0.258819f,  0.0f,  0.965926f,  0.0f,
-0.965926f, -4.37114e-08f, -0.258819f,  0.5f,
4.2222e-08f, -1.0f,  1.13133e-08f,  0.0f,
0.0f,        0.0f,        0.0f,         1.0f;

//sequence 2
estimation     << 1.91069e-15f,    4.37114e-08f,  -1.0f,     2.5f,
1.0f,    -4.37114e-08f,   0.0f,    -4.5f,
-4.37114e-08f, -1.0f,    -4.37114e-08f,  0.0f,
0.0f,          0.0f,          0.0f,          1.0f;
//sequence 2 con mapa_estanteria_bodega
estimation     << 0.998004f,    -0.0496079f,  -0.039129f,     0.17813f,
0.0480109f,    0.998016f,   -0.0407492f,    -0.0370797f,
0.0410728f, 0.0387893f,    0.998404f,  -0.0502154f,
0.0f,          0.0f,          0.0f,          1.0f;
//sequence 2 con mapa_taller
estimation     << 0.983426f,    0.0869602f,  0.159104f,     0.0537242f,
-0.0888345f,    0.996037f,   0.00469247f,    0.20925f,
-0.158066f, -0.0187486f,    0.987251f,  0.351872f,
0.0f,          0.0f,          0.0f,          1.0f;
//secuencia_prueba_estimacion_lineal con mapa_cuarto_3
estimation     << -5.73206e-15f, -4.37114e-08f, 1.0f, -0.1f,
-1.0f, 4.37114e-08f, -3.82137e-15f,   1.0f,
```

```
-4.37114e-08f, -1.0f,     -4.37114e-08f,   0.2f,
0.0f,          0.0f,          0.0f,          1.0f;*/
//secuencia_cuarto_inversa con mapa_inverso_cuarto
estimation     << 1.13133e-08f, 0.258819f, 0.965926f, 0.0f,
-1.0f, 1.28155e-07f, -2.26267e-08f,   -1.2f,
-1.29645e-07f, -0.965926f,     0.258819f,   0.8f,
0.0f,          0.0f,          0.0f,          1.0f;

tf2::Transform estimacion = EigenMatrixToTransform(estimation);
printEigenMatrix4f (TransformToEigenMatrix(estimacion));
    geometry_msgs::PoseStamped pose_stamped;
visualization_msgs::Marker marker;
visualization_msgs::Marker marker2;
visualization_msgs::MarkerArray marker_array;


//================================================================================
//  Main bucle
    //================================================================================
==
while (ros::ok() && more_data){

output_map.header.stamp = ros::Time::now();
        output_map.header.frame_id = "map";
    publisher_map.publish(output_map);


//---------------------------------------------------------------------
//  Parse one line from 'clouds_data.txt' file
//---------------------------------------------------------------------
if (next_cld){
if (cld_counter > 0){
if (cld_file.is_open()){
std::string tokem;
//Getting path
std::getline (cld_file, tokem, ' ');
cld_msg.setPath (tokem);
//Getting time
std::getline (cld_file, tokem, '\n');
cld_msg.setTime (std::atoi (tokem.c_str()));
//Prevent from loading next data
next_cld = false;
}else{
PRINT_ERROR ("File 'clouds_data.txt' lost");
more_data = false;
}
}
else{
next_cld = false;
}
}

//---------------------------------------------------------------------
//  Cloud processing
//---------------------------------------------------------------------
if (cld_msg.getTime() == simulation_time){
//Processing it as if it was a real time PointCloud message
```

```
Cloud frame;
if (frame.loadCloud (path_clds + cld_msg.getPath())){
PRINT_DEBUG ("------------------------------------------------------------------------");
PRINT_DEBUG ("Cloud '" << cld_msg.getPath() << "' loaded succesfully");
//Start time measurement
cld_timer.start();
//Downsample cloud

frame.downsample(0.04);
frame.computeNormals();

//frame.removeOutliersStatisticaly (50, 3);
//frame.removeOutliersRadius (0.06, 4);


//Transform


//Locate cloud in map
tf2::Vector3 next_position;
tf2::Vector3 actual_position;
tf2::Vector3 distance;
tf2::Matrix3x3 rotation;
float distance_scalar;
float cloud_velocity;
geometry_msgs::Point p;


tf2::Transform new_location = map.locate (frame, estimacion);
actual_position = new_location.getOrigin();
if(cld_msg.getPath() == "cloud_1.pcd"){
last_position = actual_position;
}

distance = operator-(actual_position, last_position);
next_position = operator+(distance, actual_position);

rotation = new_location.getBasis();
estimacion.setBasis(rotation);
estimacion.setOrigin(next_position);
last_position = actual_position;



pose_stamped.header.frame_id = "map";
pose_stamped.header.stamp = ros::Time::now();
marker.header.frame_id = "map";
marker.header.stamp = ros::Time::now();
marker.ns = "markers";
marker.action = visualization_msgs::Marker::ADD;
marker.pose.orientation.w = 1.0;
marker.id = 0;
marker.type = visualization_msgs::Marker::POINTS;
marker.scale.x = 0.05;
marker.scale.y = 0.05;
marker.color.g = 1.0f;
marker.color.a = 1.0;
```

```
marker2.header.frame_id = "map";
marker2.header.stamp = ros::Time::now();
marker2.ns = "markers";
marker2.action = visualization_msgs::Marker::ADD;
marker2.pose.orientation.w = 1.0;
marker2.id = 1;
marker2.type = visualization_msgs::Marker::POINTS;
marker2.scale.x = 0.05;
marker2.scale.y = 0.05;
marker2.color.r = 1.0f;
marker2.color.a = 1.0;

pose_stamped.pose.orientation.x = new_location.getRotation().getX();
pose_stamped.pose.orientation.y = new_location.getRotation().getY();
pose_stamped.pose.orientation.z = new_location.getRotation().getZ();
pose_stamped.pose.orientation.w = new_location.getRotation().getW();

p.x = pose_stamped.pose.position.x = new_location.getOrigin().getX();
p.y = pose_stamped.pose.position.y = new_location.getOrigin().getY();
p.z = pose_stamped.pose.position.z = new_location.getOrigin().getZ();

if(map.p2plane){
marker2.points.push_back(p);
}
else{
marker.points.push_back(p);
}

drone.setLocation (new_location);
//Calculate velocity from MAP movement
tf2::Vector3 velocity = drone.getVelocityEstimation(cld_msg.getTime() - time_last_cloud);
time_last_cloud = cld_msg.getTime();
//End time measurement
double t_cld = cld_timer.stop();
if(t_cld<cld_min) cld_min = t_cld;
if(t_cld>cld_max) cld_max = t_cld;
PRINT_DEBUG ("t_cld = " << t_cld);
estimacion = new_location;
previous_timer = process.timer.step();
}
else{
PRINT_ERROR ("Unable to load '" << cld_msg.getPath() << "'");
}
//Ask for a new PointCloud message
next_cld = true;
cld_counter --;
process.step();
//Pause simulation
if (LOG_LEVEL == 0 && stop_cld){
//Debug info
PRINT_DEBUG (cld_msg);
PRINT_DEBUG (PRINT_TRANSFORM (drone.getLocationEstimation()));
PRINT_DEBUG ("-----------------------------------------------------------------");
//Wait user
std::cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
}
```

```
pcl::toROSMsg(*(map.output_cloud.point_cloud),output_cloud);
output_cloud.header.stamp = ros::Time::now();
output_cloud.header.frame_id = "map";
    publisher_cloud.publish(output_cloud);
pub_transform.publish(pose_stamped);
marker_pub.publish(marker);
marker_pub2.publish(marker2);


}
//-------------------------------------------------------------------
//  Print simulation progress
//-------------------------------------------------------------------
if (next_cld || cld_counter == 0 ){
//Simulation progress is only displayed every time a PointCloud is processed
//and also when all data has been processed (100%)
PRINT_INFO ("       [" << std::setfill(' ') << std::setw(3) << std::right << process.progress()
<< "%]" << std::setw(10) << std::right << std::fixed << std::setprecision(3)
<< (float)simulation_time/1000 << " s" << std::setw(10) << std::right << std::fixed
<< std::setprecision(3) << process.lastTime() << " s"  << std::setw(16) << std::right
<< std::fixed << std::setprecision(3) << process.timeRemainingEstimation() << " s" );
}

//-------------------------------------------------------------------
//  Stops the simulation when data ends
//-------------------------------------------------------------------

if (cld_counter == 0 ){
more_data = false;
}

simulation_time ++;
}
map.saveError();
double cld_media = cld_timer.totalTime() / (double)cld_counter_BU;
printf("Average CLD processing time: %f\n",cld_media);
printf("Max     CLD processing time: %f\n",cld_max);
printf("Min     CLD processing time: %f\n",cld_min);

//=====================================================================
//  Saving results
   //=====================================================================
==
   drone.drawTrayectory(map);
   map.saveMap (main_path + "/sequence_" + sequence + "result.pcd");

//=====================================================================
//  The processing has successfully ended. Shutting down the node
   //=====================================================================
==
   if (!sigint_called){
PRINT_INFO ("");
PRINT_INFO
("=====================================================================");
PRINT_INFO ("|                                                                   |");
PRINT_INFO ("|                    TFG_PROCESSOR                                  |");
PRINT_INFO ("|                                                                   |");
```

```
PRINT_INFO
("|=========================================================================|");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("| Author:              David Turbica, Ramiro Costa                        |");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("| Description:      ICP-based RGBd camera location system on a            |");
PRINT_INFO ("|                   known environment                                     |");
PRINT_INFO ("|                                                                         |");
PRINT_INFO
("|=========================================================================|");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("| Node terminated:  The processing has successfully ended.                |");
PRINT_INFO ("|                                                                         |");
PRINT_INFO
("===========================================================================");
}
//Shutting down the node
ros::shutdown();
return 0;
}

void customSigintHandler(int sig){
//---------------------------------------------------------------------------
// Function that overrides default ROS sigint handler. It allows to print
//  some information when the node has been shuted down and end child
//  processes if needed.
//---------------------------------------------------------------------------
//Close files
if (origin_file.is_open())  origin_file.close();
if (cld_file.is_open())     cld_file.close();
//Stop debugging stpe-by-step
stop_cld = false;
//Advice a sigint has been called
sigint_called = true;
//Print end message
PRINT_INFO ("");
PRINT_INFO
("===========================================================================");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("|                        TFG_PROCESSOR                                    |");
PRINT_INFO ("|                                                                         |");
PRINT_INFO
("|=========================================================================|");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("| Author:              David Turbica, Ramiro Costa                        |");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("| Description:      ICP-based RGBd camera location system on a            |");
PRINT_INFO ("|                   known environment                                     |");
PRINT_INFO ("|                                                                         |");
PRINT_INFO
("|=========================================================================|");
PRINT_INFO ("|                                                                         |");
PRINT_INFO ("| Node terminated:  No reason given (Ctrl+c)                    |");
PRINT_INFO ("|                                                          |");
PRINT_INFO
("===========================================================================");
//Overrides ROS default Sigint Handler
```

```
ros::shutdown();
}

void printHelp (char* program_name){
std::cout << std::endl;
std::cout << "Usage: " << program_name << " <option(s)>\n"
  << "Options:\n"
  << "        -h,--help                    Show help\n"
  << "        -d,--debug                   Set Verbose level to DEBUG\n"
  << "        -i,--info                    Set Verbose level to INFO\n"
  << "        -e,--error                   Set Verbose level to ERROR\n"
  << "        -di,--debugimu               DEBUG + step-by-step IMU\n"
  << "        -dc,--debugcloud             DEBUG + step-by-step CLOUD\n"
  << "        -db,--debugboth              DEBUG + step-by-step IMU + CLOUD\n"
  << "        -n,--nolog                   Set Verbose level to NOLOG\n"
  << "        -m <path>,--map <path>           Set the path to a custom map\n"
  << "        -s <number>, --seq <number>      Sets the sequence to be processed\n"
  << "        -f <number>, --filter <number>   Sets the size of the imu filter\n"
  << "Default settings:\n"
  << "        -Verbose level:              DEBUG\n"
  << "        -Step-by-step: No IMU, no CLOUD\n"
  << "        -Map:                        src/tfg_v2/map/\n"
  << "        -Sequence:                   1 (sequence_1)\n"
  << "        -Filter size:                20 (max 50)\n" << std::endl;
}
```

Anexo 23. tfg_Timer.h

```
//================================================================
//
// File: tfg_Timer.h
//  Author: David Turbica
//
//================================================================

#ifndef __TFG_TIMER_H_INCLUDED__
#define __TFG_TIMER_H_INCLUDED__


//C++ includes
#include <ctime>

//ROS includes


//PCL includes


//TFG includes


namespace tfg {

class Timer{
//Variables
private:
double time;
clock_t ticks;
clock_t ticks_period;
bool started;

//Functions
public:
Timer (void);
double totalTime (void);
void start (void);
double step (void);
double period (void);
double stop (void);
};

} //end of namespace tfg

#endif //__TFG_TIMER_H_INCLUDED
```

Anexo 24. tfg_Timer.cpp

```
//===========================================================================
//
// File: tfg_Timer.cpp
//  Author: David Turbica
//
//===========================================================================

#include "tfg_processor/tfg_Timer.h"

namespace tfg {

Timer::Timer (void){
//---------------------------------------------------------------------------
// Default constructor.
//---------------------------------------------------------------------------
this->time = 0.0;
this->started = false;
}

double Timer::totalTime (void){
//---------------------------------------------------------------------------
// Returns its 'time' value.
//---------------------------------------------------------------------------
return this->time;
}

void Timer::start (void){
//---------------------------------------------------------------------------
// Starts meassuring time.
//---------------------------------------------------------------------------
this->started = true;
this->ticks = clock();
this->ticks_period = this->ticks;
}

double Timer::step (void){
//---------------------------------------------------------------------------
// Returns the time elapsed since last 'start()' was called. Time is not
//  added to the total time.
//---------------------------------------------------------------------------
double time_elapsed = 0;
if (this->started){
time_elapsed = (double)(clock()-this->ticks) / CLOCKS_PER_SEC;
this->ticks_period = clock();
}
return time_elapsed;
}

double Timer::period (void){
//---------------------------------------------------------------------------
// Returns the time elapsed since last 'start()' or 'step()' or 'period()'
//  was called. No time is added to total time.
//---------------------------------------------------------------------------
double time_elapsed = 0;
```

```
if (this->started){
time_elapsed = (double)(clock()-this->ticks_period) / CLOCKS_PER_SEC;
this->ticks_period = clock();
}
return time_elapsed;
}

double Timer::stop (void){
//------------------------------------------------------------------------
// Returns the time elapsed since last 'start()' was called and adds it to
//  the total time.
//------------------------------------------------------------------------
double time_elapsed = 0;
if (this->started){
time_elapsed = (double)(clock()-this->ticks) / CLOCKS_PER_SEC;
this->time += time_elapsed;
this->started = false;
}
return time_elapsed;
}

} //end of namespace tfg
```

Anexo 25. tfg_utilities.h

```
//=================================================================
//
// File: tfg_utilities.h
//  Author:
//
//=================================================================

#ifndef __TFG_UTILITIES_H_INCLUDED__
#define __TFG_UTILITIES_H_INCLUDED__

#include <ros/ros.h>

//C++ includes
#include <stdio.h>
#include <string>
#include <math.h>
#include <Eigen/Dense>
#include <tf2/LinearMath/Vector3.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <tf2/LinearMath/Transform.h>

//ROS includes


//PCL includes
#include <pcl/common/transforms.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>

//TFG includes
#include "tfg_Logger.h"

//Macros
#define PRINT_VECTOR3(vector3) vector3.getX() << "    " << vector3.getY() << "    " <<
vector3.getZ()

#define PRINT_TRANSFORM(transform) std::setw(14) << std::right << std::fixed <<
std::setprecision(5) << transform.getBasis()[0][0]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[0][1]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[0][2]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getOrigin()[0]\
<< std::endl\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[1][0]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[1][1]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[1][2]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getOrigin()[1]\
<< std::endl\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[2][0]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[2][1]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getBasis()[2][2]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << transform.getOrigin()[2]\
<< std::endl\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0\
```

```
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 0.0\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << 1.0

#define PRINT_MATRIX(matrix)       std::setw(14) << std::right << std::fixed <<
std::setprecision(5) << matrix[0][0]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[0][1]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[0][2]\
<< std::endl\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[1][0]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[1][1]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[1][2]\
<< std::endl\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[2][0]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[2][1]\
<< std::setw(14) << std::right << std::fixed << std::setprecision(5) << matrix[2][2]



double round (double number, int decimals);
tf2::Vector3 roundVector3 (tf2::Vector3 vector, int decimals);
void printEigenMatrix4f (const Eigen::Matrix4f & matrix);
Eigen::Matrix4f TransformToEigenMatrix (tf2::Transform input);
tf2::Transform EigenMatrixToTransform (Eigen::Matrix4f input);
Eigen::Matrix4f eulerTransform(float a, float b, float c, float x, float y, float z);

#endif //__TFG_UTILITIES_H_INCLUDED
```

Anexo 26. tfg_utilities.cpp

```
//=============================================================================
//
// File: tfg_utilities.cpp
//  Author: David Turbica
//
//=============================================================================

#include "tfg_processor/tfg_utilities.h"

double round (double number, int decimals){
double factor = pow(10.0,(double)decimals);
return (double)((int)(number*factor))/factor;
}

tf2::Vector3 roundVector3 (tf2::Vector3 vector, int decimals){
return tf2::Vector3 (round(vector.getX(),decimals), round(vector.getY(),decimals),
round(vector.getZ(),decimals));
}

void printEigenMatrix4f (const Eigen::Matrix4f & matrix){
printf ("Rotation matrix :\n");
printf ("    | %6.3f %6.3f %6.3f | \n", matrix (0, 0), matrix (0, 1), matrix (0, 2));
printf ("R = | %6.3f %6.3f %6.3f | \n", matrix (1, 0), matrix (1, 1), matrix (1, 2));
printf ("    | %6.3f %6.3f %6.3f | \n", matrix (2, 0), matrix (2, 1), matrix (2, 2));
printf ("Translation vector :\n");
printf ("t = < %6.3f, %6.3f, %6.3f >\n\n", matrix (0, 3), matrix (1, 3), matrix (2, 3));
}

Eigen::Matrix4f TransformToEigenMatrix (tf2::Transform input){
Eigen::Matrix4f output;
output(0,0) = input.getBasis()[0][0];
output(0,1) = input.getBasis()[0][1];
output(0,2) = input.getBasis()[0][2];
output(0,3) = input.getOrigin()[0];
output(1,0) = input.getBasis()[1][0];
output(1,1) = input.getBasis()[1][1];
output(1,2) = input.getBasis()[1][2];
output(1,3) = input.getOrigin()[1];
output(2,0) = input.getBasis()[2][0];
output(2,1) = input.getBasis()[2][1];
output(2,2) = input.getBasis()[2][2];
output(2,3) = input.getOrigin()[2];
output(3,0) = 0.0;
output(3,1) = 0.0;
output(3,2) = 0.0;
output(3,3) = 1.0;
return output;
}

tf2::Transform EigenMatrixToTransform (Eigen::Matrix4f input){
tf2::Matrix3x3 rotation (input(0,0), input(0,1), input(0,2),
input(1,0), input(1,1), input(1,2),
input(2,0), input(2,1), input(2,2));
tf2::Vector3 position (input(0,3), input(1,3), input(2,3));
```

```
return tf2::Transform (rotation, position);
}

Eigen::Matrix4f eulerTransform(float a, float b, float c, float x, float y, float z){
Eigen::Matrix4f transformation_matrix = Eigen::Matrix4f::Identity ();
// A rotation matrix
transformation_matrix (0, 0) = cos(a)*cos(b)*cos(c)-sin(a)*sin(c);
transformation_matrix (0, 1) = -cos(a)*cos(b)*sin(c)-sin(a)*cos(c);
transformation_matrix (0, 2) = cos(a)*sin(b);
transformation_matrix (1, 0) = sin(a)*cos(b)*cos(c)+cos(a)*sin(c);
transformation_matrix (1, 1) = -sin(a)*cos(b)*sin(c)+cos(a)*cos(c);
transformation_matrix (1, 2) = sin(a)*cos(b);
transformation_matrix (2, 0) = -sin(b)*cos(c);
transformation_matrix (2, 1) = sin(b)*sin(c);
transformation_matrix (2, 2) = cos(b);
// Translation on X,Y,Z axis (in meters)
transformation_matrix (0, 3) = x;
transformation_matrix (1, 3) = y;
transformation_matrix (2, 3) = z;
// Perspective vector
transformation_matrix (3, 0) = 0.0;
transformation_matrix (3, 1) = 0.0;
transformation_matrix (3, 2) = 0.0;
return transformation_matrix;
}
```